SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

P&S Processing-in-Memory
Fall 2022
10 January 2023

Nastaran Hajinazar* **Geraldo F. Oliveira***

Sven Gregorio Joao Ferreira Nika Mansouri Ghiasi

Minesh Patel Mohammed Alser Saugata Ghose

Juan Gómez–Luna Onur Mutlu

SAFARI SFU SIMON FRASER UNIVERSITY ETH zürich UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Executive Summary

- **Motivation:** Processing-using-Memory (PuM) architectures can effectively perform bulk bitwise computation
- **Problem**: Existing PuM architectures are not widely applicable
  - Support only a limited and specific set of operations
  - Lack the flexibility to support new operations
  - Require significant changes to the DRAM subarray
- **Goals**: Design a processing-using-DRAM framework that:
  - Efficiently implements complex operations
  - Provides the flexibility to support new desired operations
  - Minimally changes the DRAM architecture
- **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  1. Efficiently computing complex operations
  2. Providing the ability to implement arbitrary operations as required
  3. Using a massively-parallel in-DRAM SIMD substrate
- **Key Results**: SIMDRAM provides:
  - 88x and 5.8x the throughput and 257x and 31x the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
  - 21x and 2.1x the performance of the CPU and GPU over seven real-world applications

Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

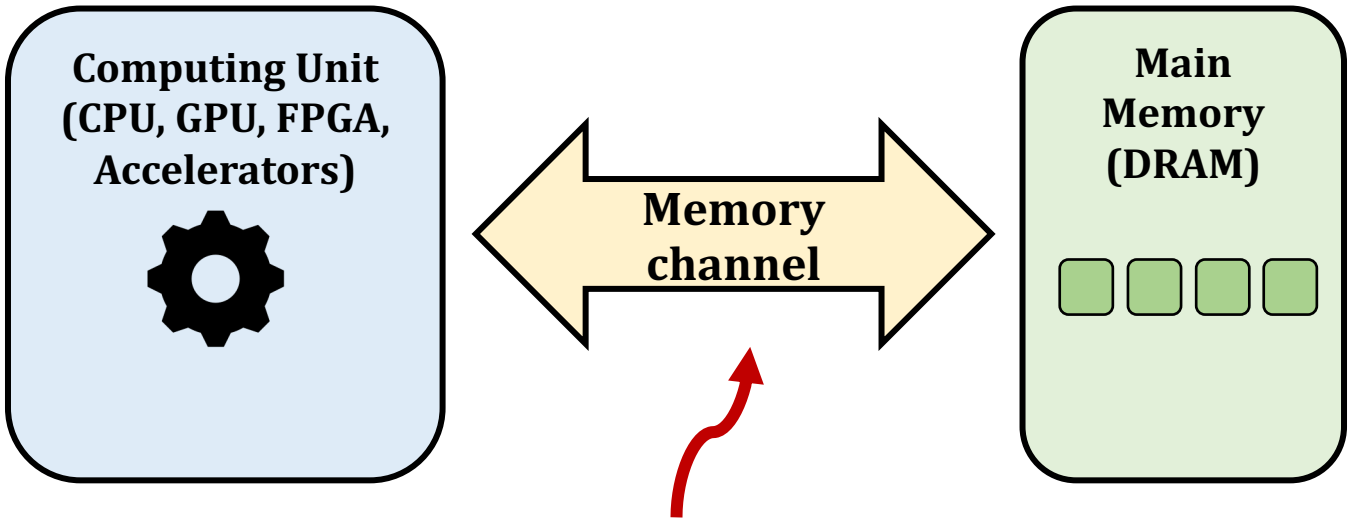
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

Data Movement Bottleneck

- Data movement is a major bottleneck

More than 60% of the total system energy is spent on data movement[1]

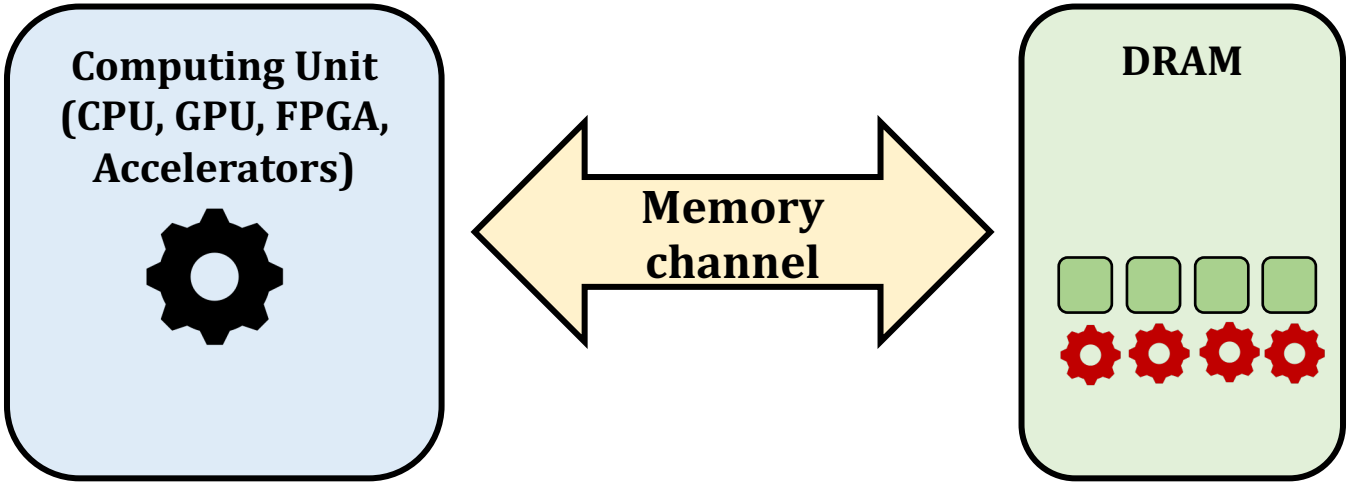


Bandwidth-limited and power-hungry memory channel

[1] A. Boroumand et al., "*Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks*," ASPLOS, 2018

Processing-in-Memory (PIM)

- **Processing-in-Memory:** moves computation closer to where the data resides
  - Reduces/eliminates the need to move data between processor and DRAM



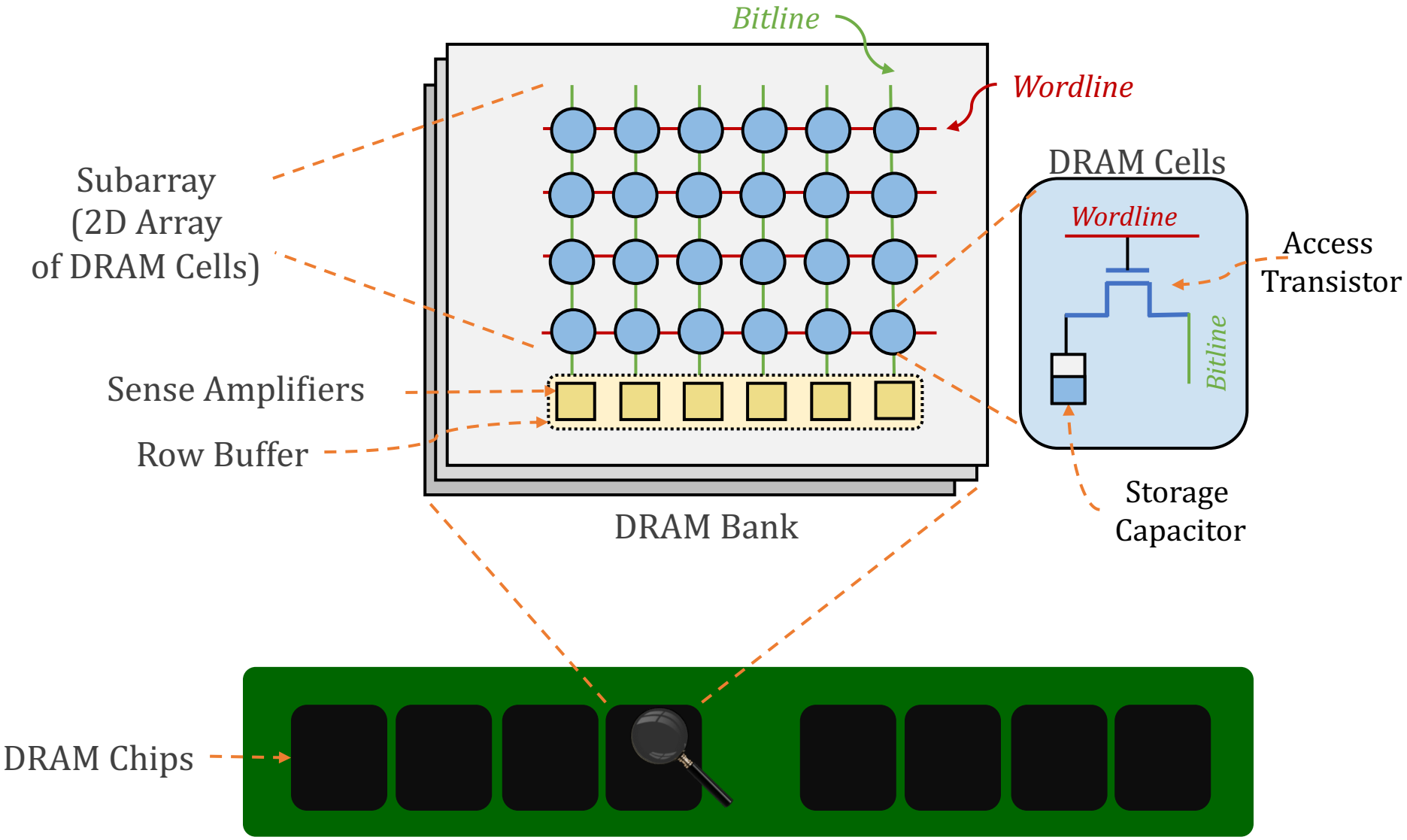
Processing-using-Memory (PuM)

- PuM: Exploits analog operation principles of the memory circuitry to perform computation
  - Leverages the large internal bandwidth and parallelism available inside the memory arrays
- A common approach for PuM architectures is to perform bulk bitwise operations
  - Simple logical operations (e.g., AND, OR, XOR)
  - More complex operations (e.g., addition, multiplication)

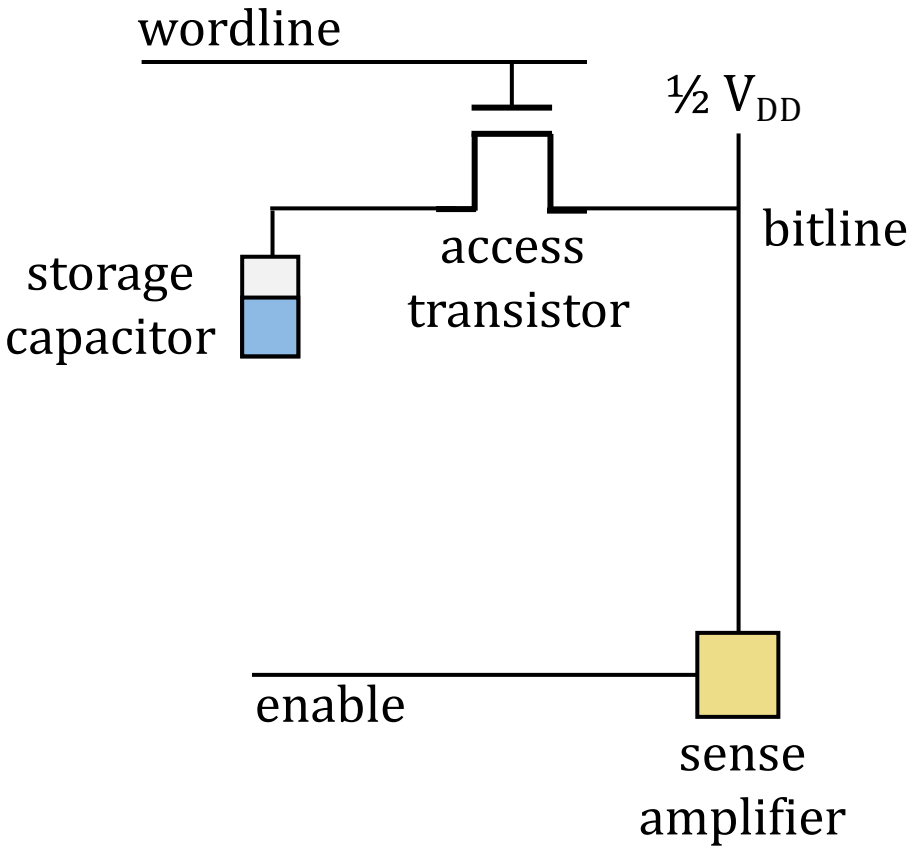
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

Inside a DRAM Chip



DRAM Cell Operation

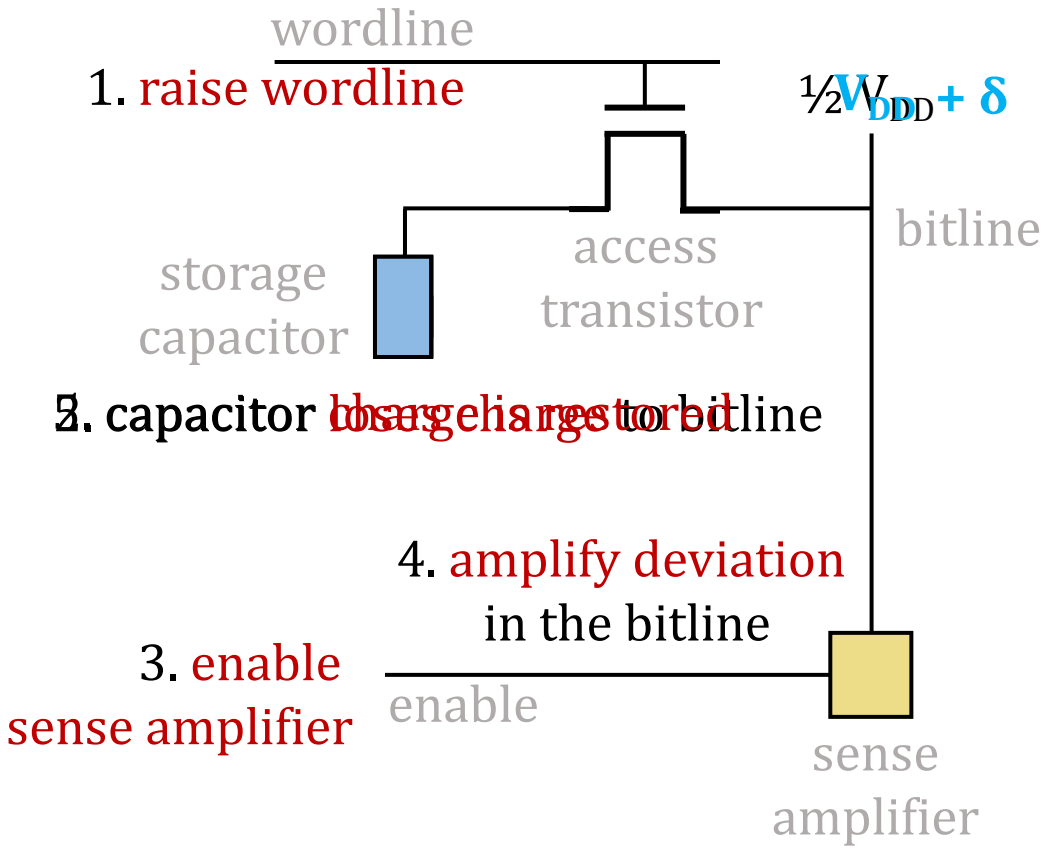


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation (1/3)

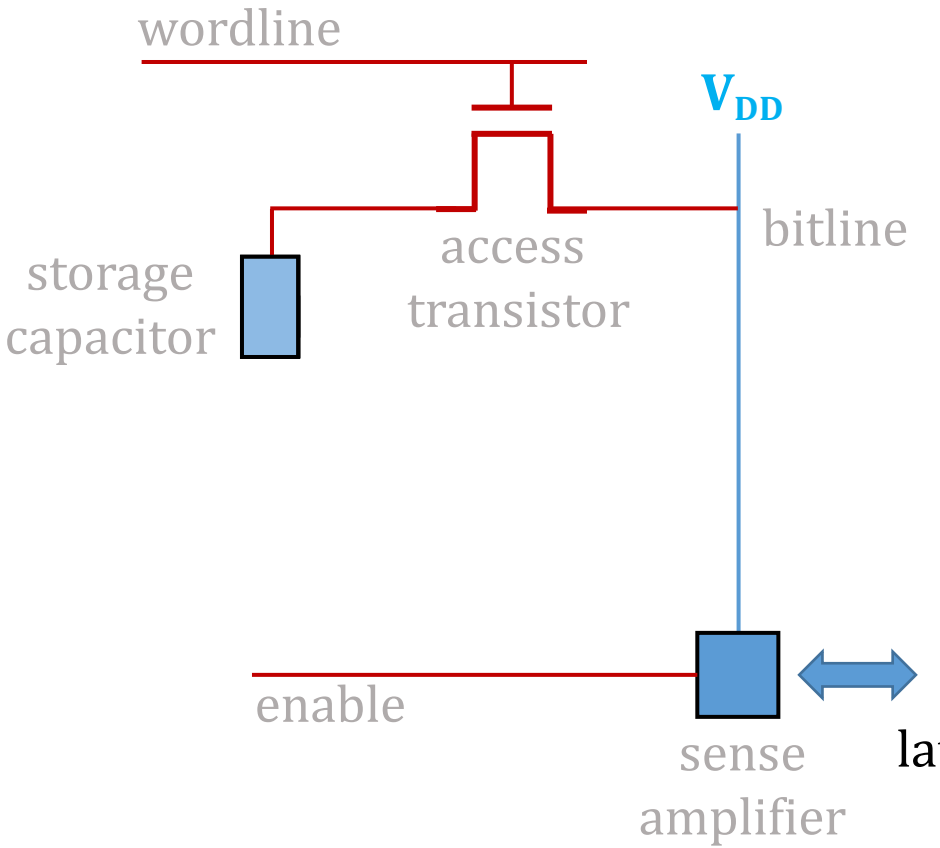


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation (2/3)



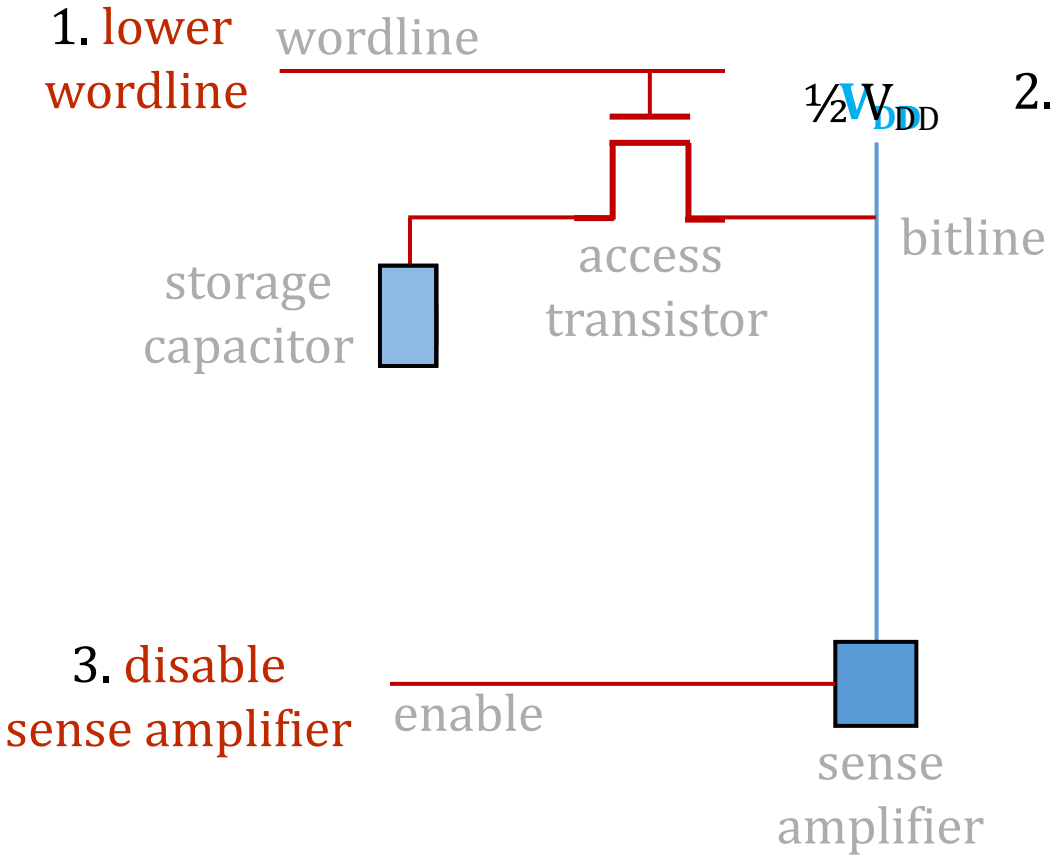
1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

read/write charge latched in sense amplifier

DRAM Cell Operation (3/3)

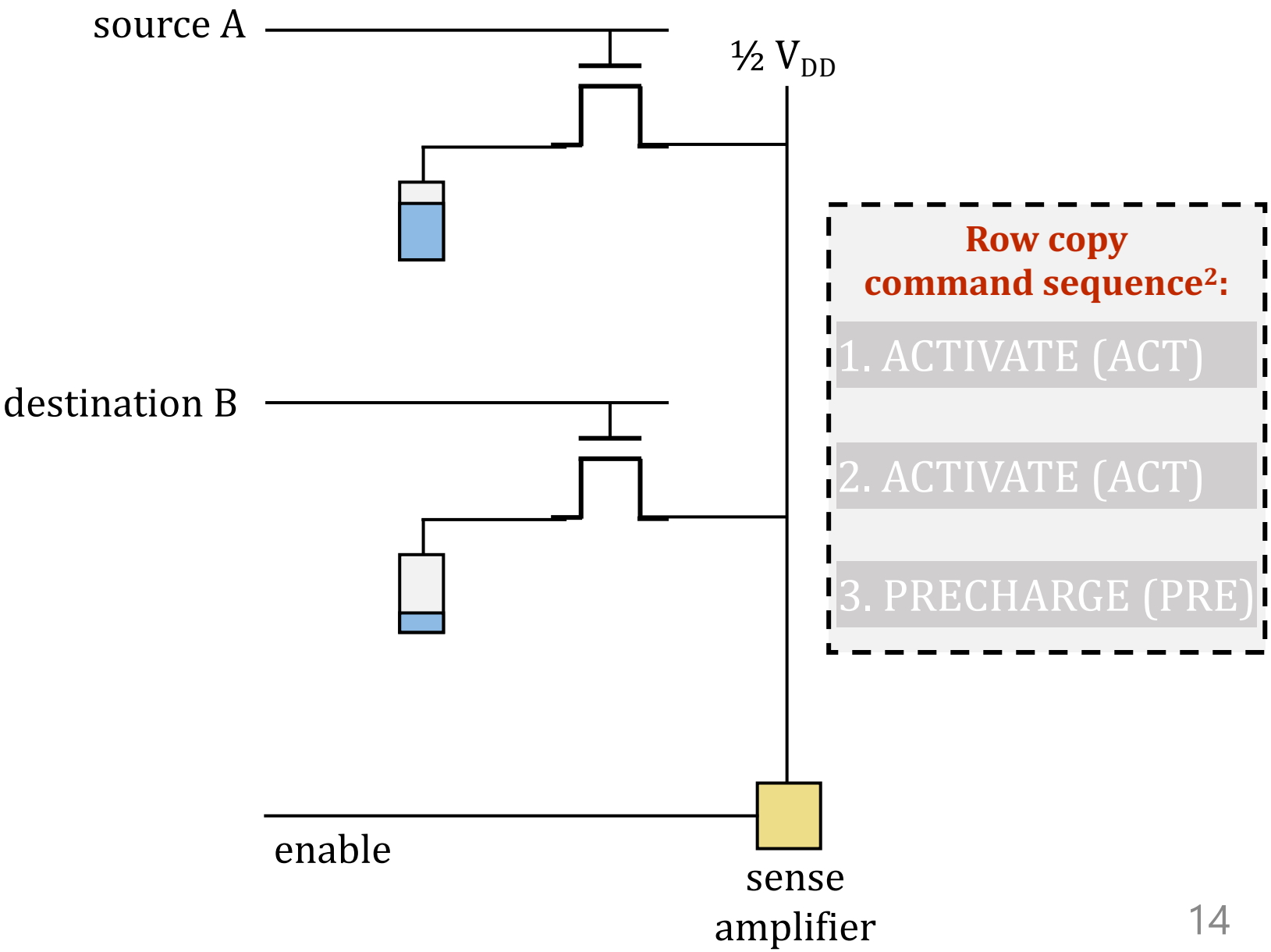


1. ACTIVATE (ACT)

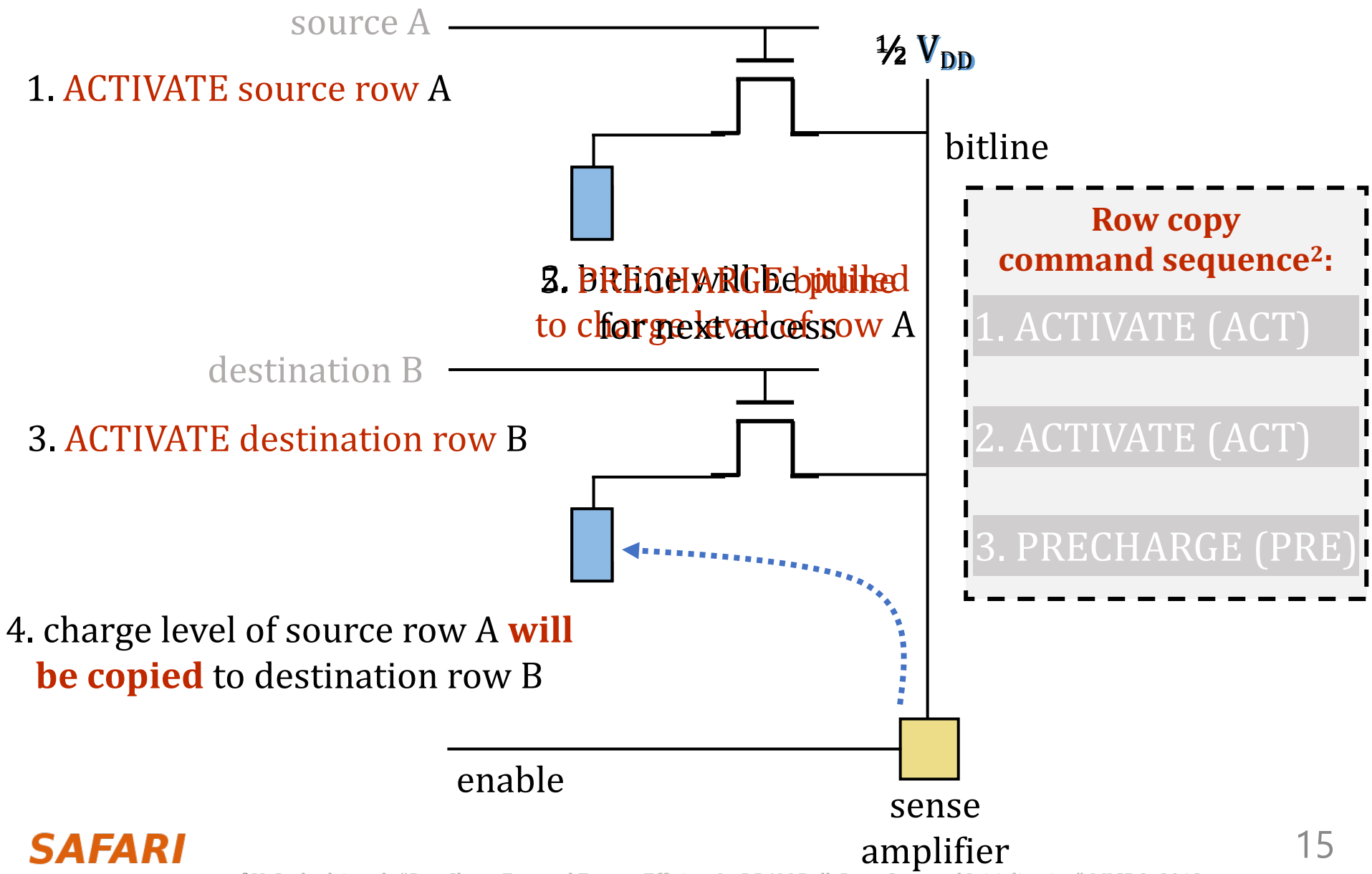
2. READ/WRITE

3. PRECHARGE (PRE)

RowClone: In-DRAM Row Copy (1/2)

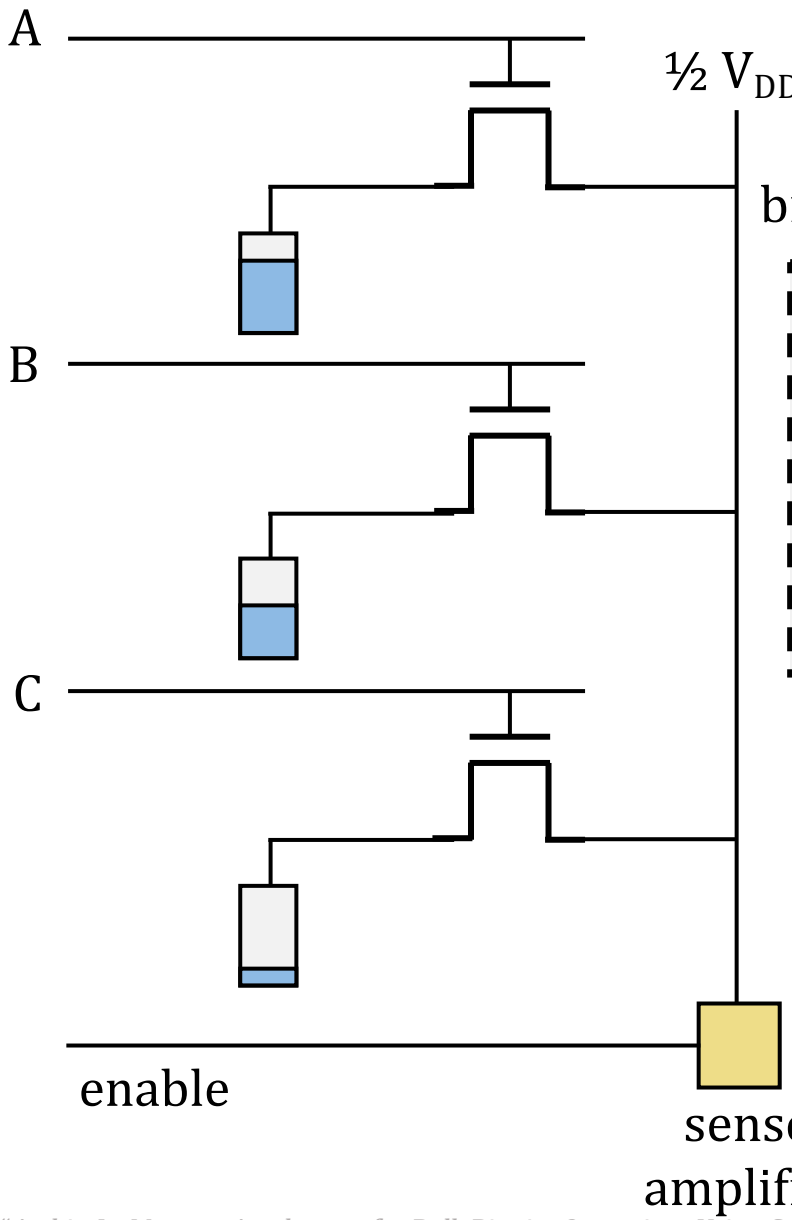


RowClone: In-DRAM Row Copy (2/2)



[2] V. Seshadri et al., *"RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"*, MICRO, 2013

Triple-Row Activation: Majority Function



Majority function command sequence[3]:

1. ACTIVATE (ACT)
2. PRECHARGE (PRE)

[3] V. Seshadri et al., *"Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology"*, MICRO, 2017

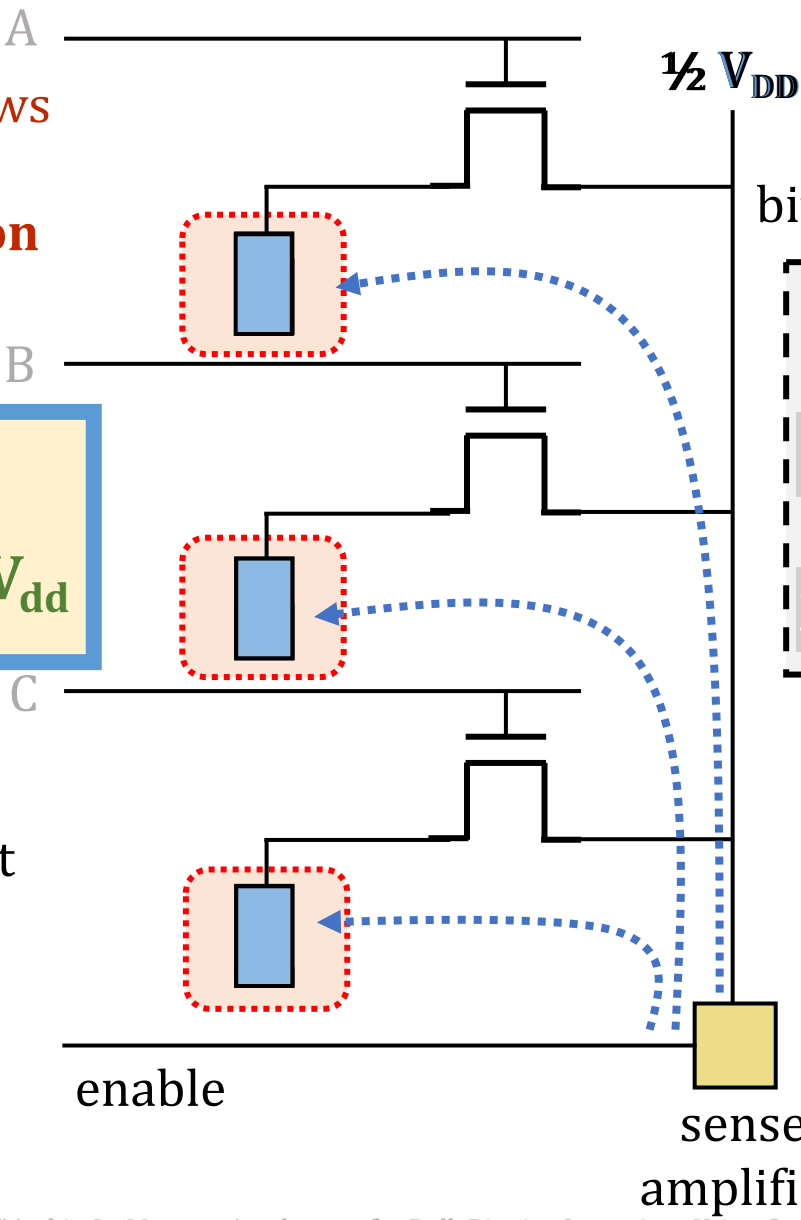
Triple-Row Activation: Majority Function

1. ACTIVATE three rows simultaneously
→ **triple-row activation**

MAJ(A, B, C) =
MAJ(V_{dd}, V_{dd}, 0) = V_{dd}

3. values in cells A, B, C will be overwritten with the majority output

4. PRECHARGE bitline for next access

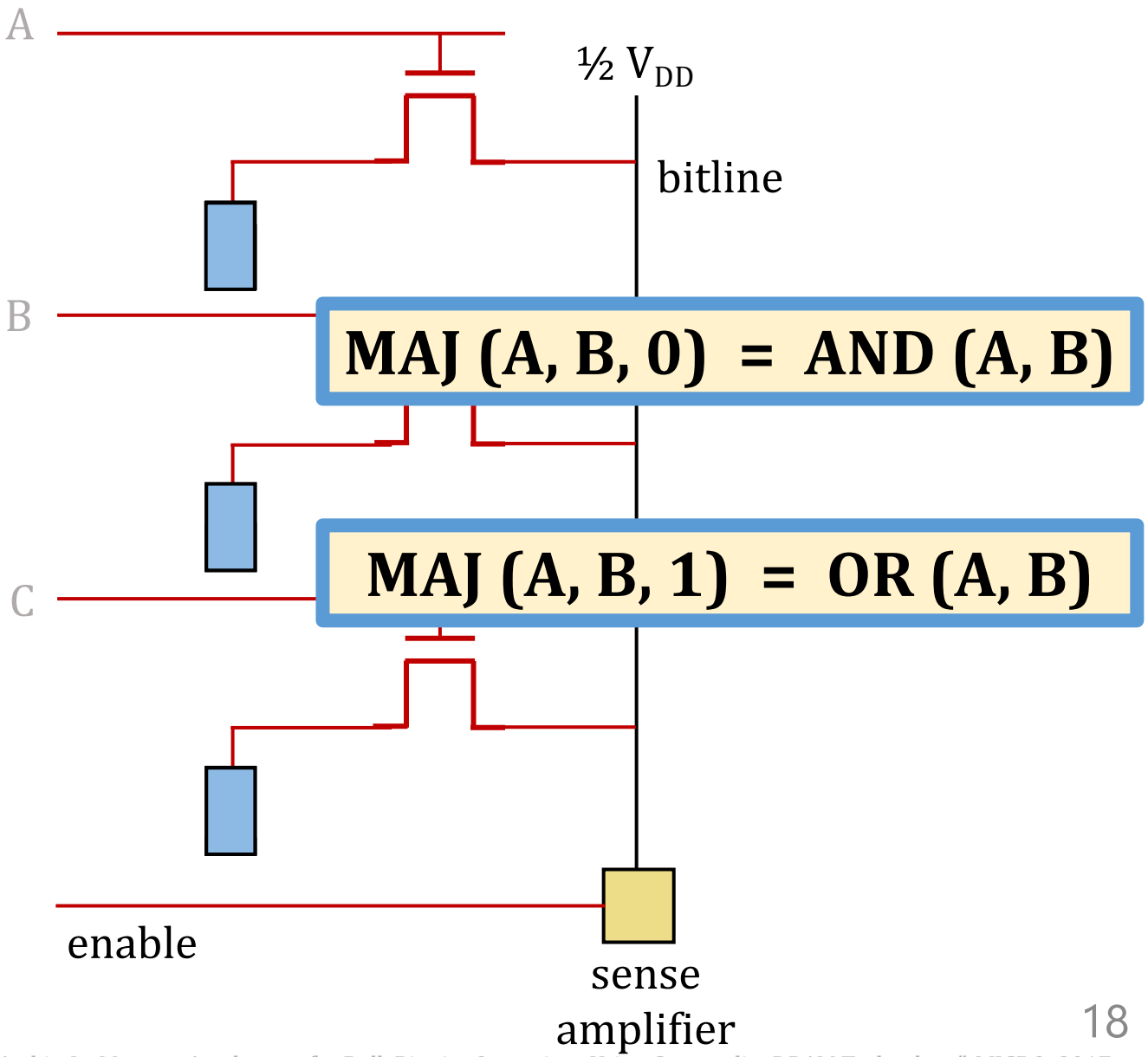


Majority function command sequence[3]:

1. ACTIVATE (ACT)
2. PRECHARGE (PRE)

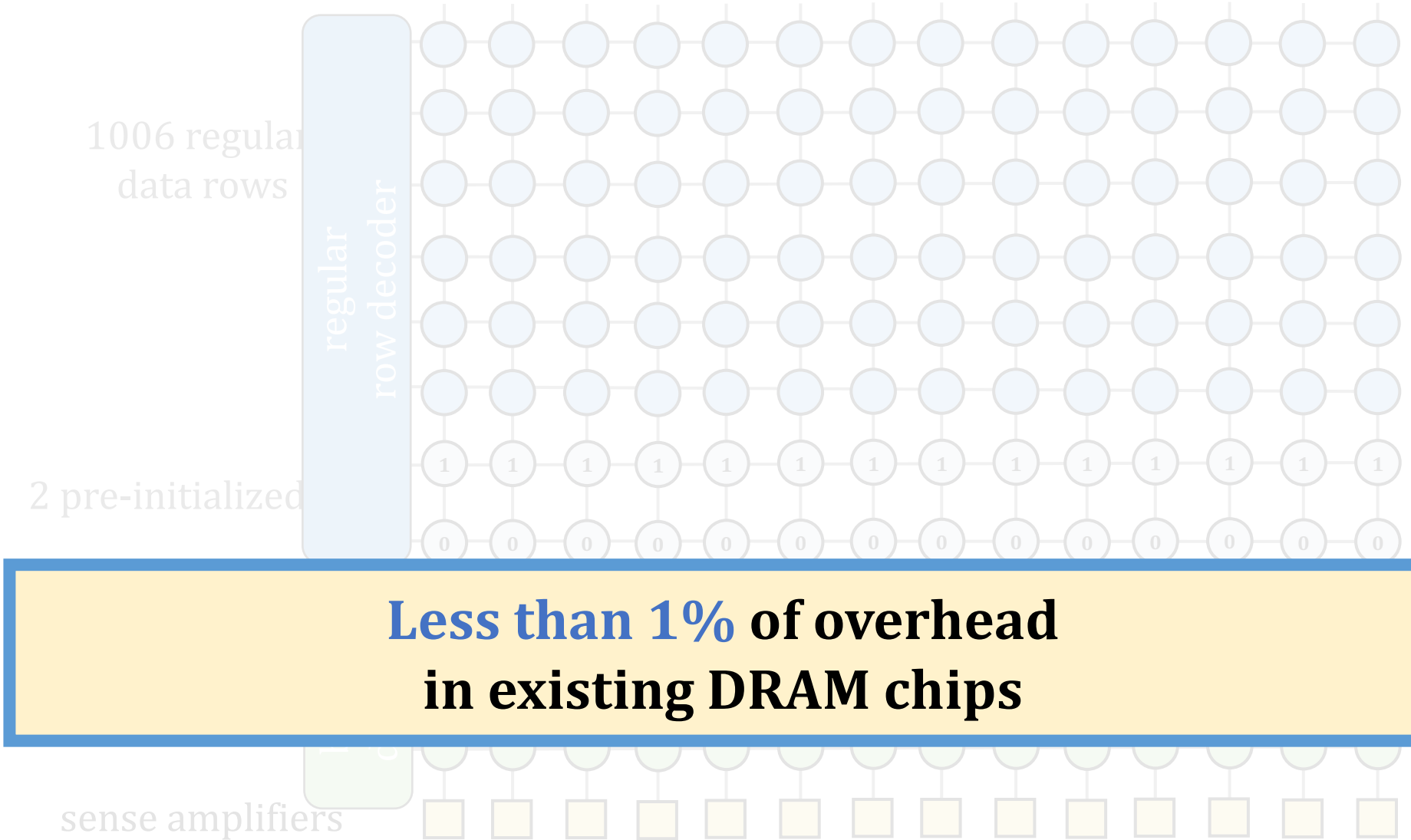
[3] V. Seshadri et al., *"Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology"*, MICRO, 2017

Ambit: In-DRAM Bulk Bitwise AND/OR



V. Seshadri et al., *"Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology"*, MICRO, 2017

Ambit: Subarray Organization



Less than 1% of overhead in existing DRAM chips

V. Seshadri et al., *"Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology"*, MICRO, 2017

PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)
  - Not widely applicable
- Support a **limited** set of operations
  - Lack the flexibility to support new operations
- Require **significant changes** to the DRAM
  - Costly (e.g., area, power)

PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

Shortcomings:

- Support **only basic** operations (e.g., Boolean operations, addition)

Need a framework that aids general adoption of PuM, by:

- Efficiently implementing complex operations

- Providing flexibility to support new operations

- Costly (e.g., area, power)

Our Goal

Goal: Design a PuM framework that

- Efficiently implements complex operations
- Provides the flexibility to support new desired operations
- Minimally changes the DRAM architecture

Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
    - Processing-using-DRAM Substrate
    - Framework
4. System Integration
5. Evaluation
6. Conclusion

Key Idea

- **SIMDRAM:** An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  - Efficiently computing complex operations in DRAM
  - Providing the ability to implement arbitrary operations as required
  - Using an in-DRAM massively-parallel SIMD substrate that requires minimal changes to DRAM architecture

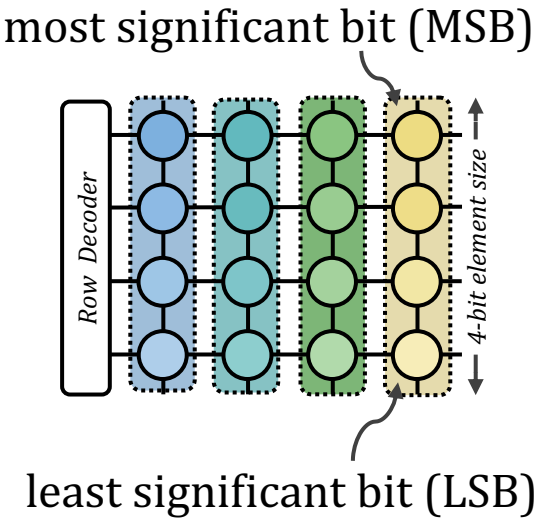
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

SIMDRAM: PuM Substrate

- SIMDRAM framework is built around a DRAM substrate that enables two techniques:

(1) Vertical data layout

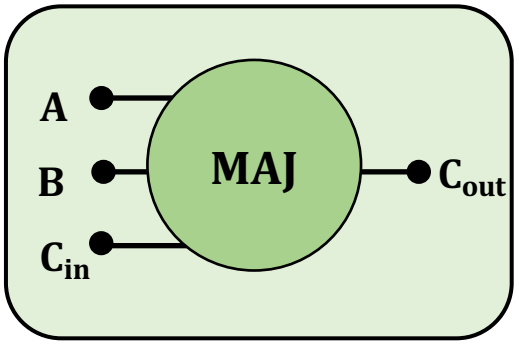


Pros compared to the conventional horizontal layout:

- Implicit shift operation
- Massive parallelism

(2) Majority-based computation

$C_{out} = AB + AC_{in} + BC_{in}$



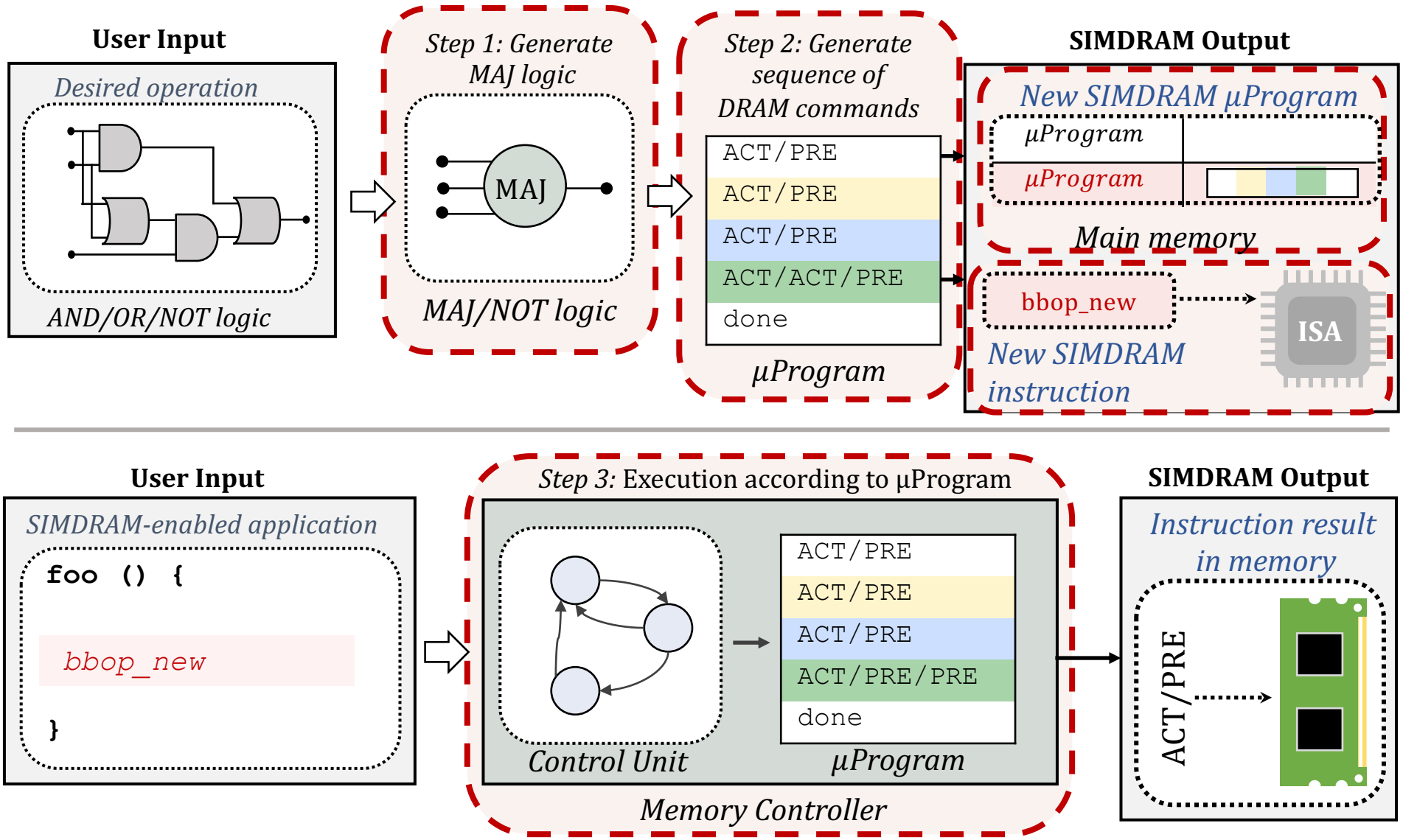
Pros compared to AND/OR/NOT-based computation:

- Higher performance
- Higher throughput
- Lower energy consumption

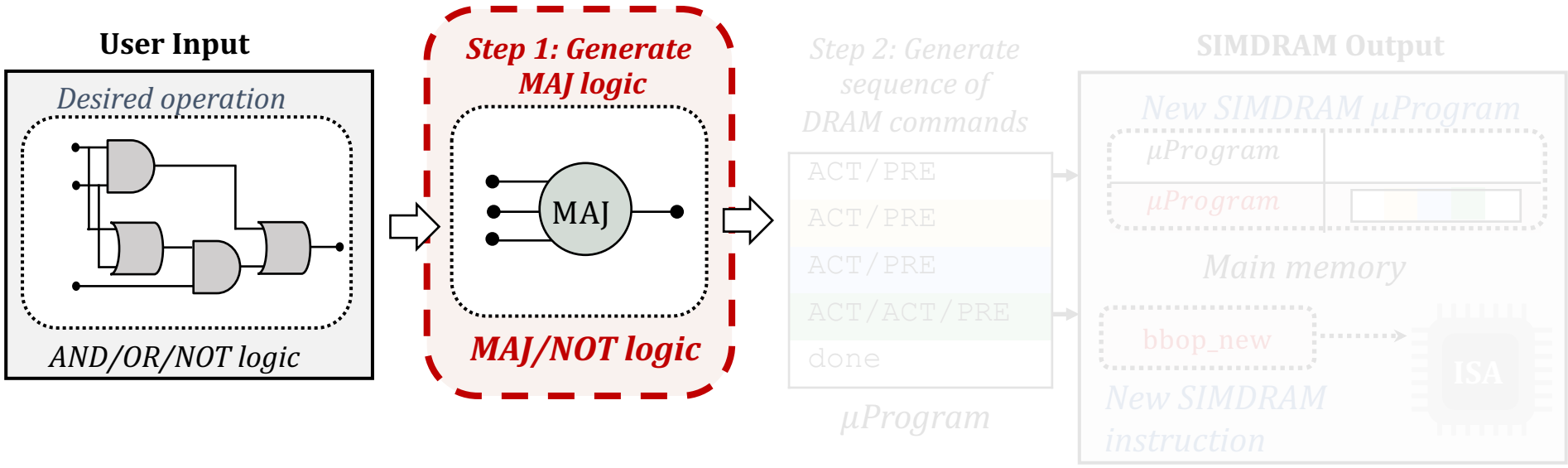
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

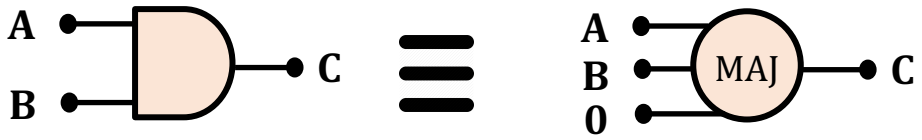
SIMDRAM Framework



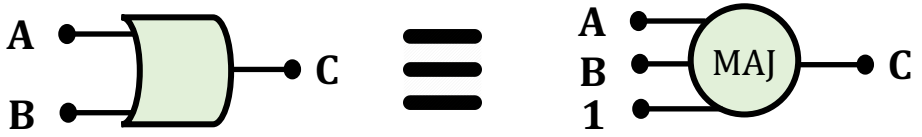
SIMDRAM Framework: Step 1



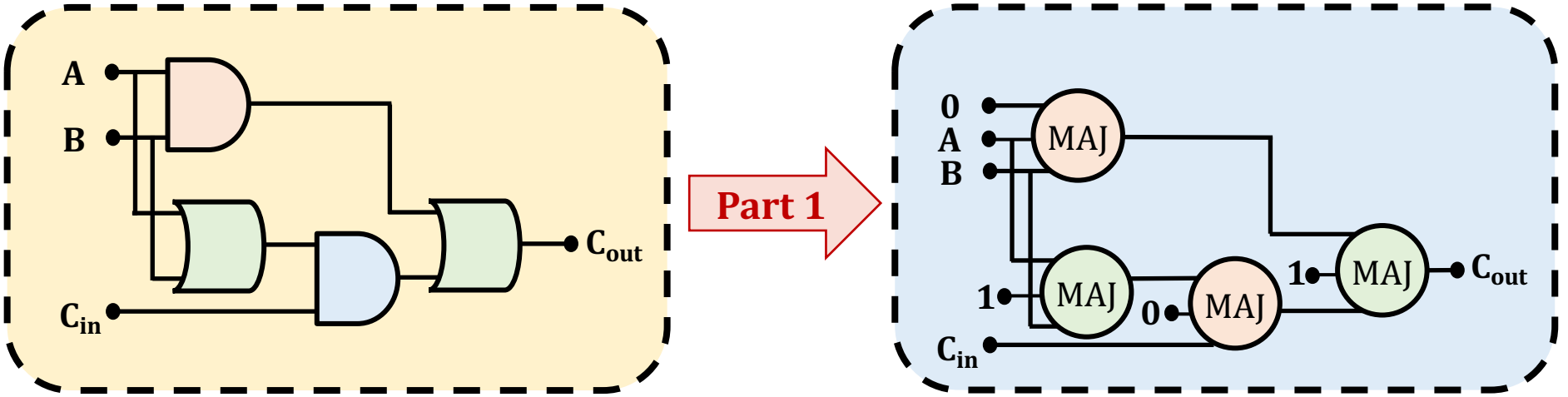
Step 1: Naïve MAJ/NOT Implementation



output is “1” only when A = B = “1”

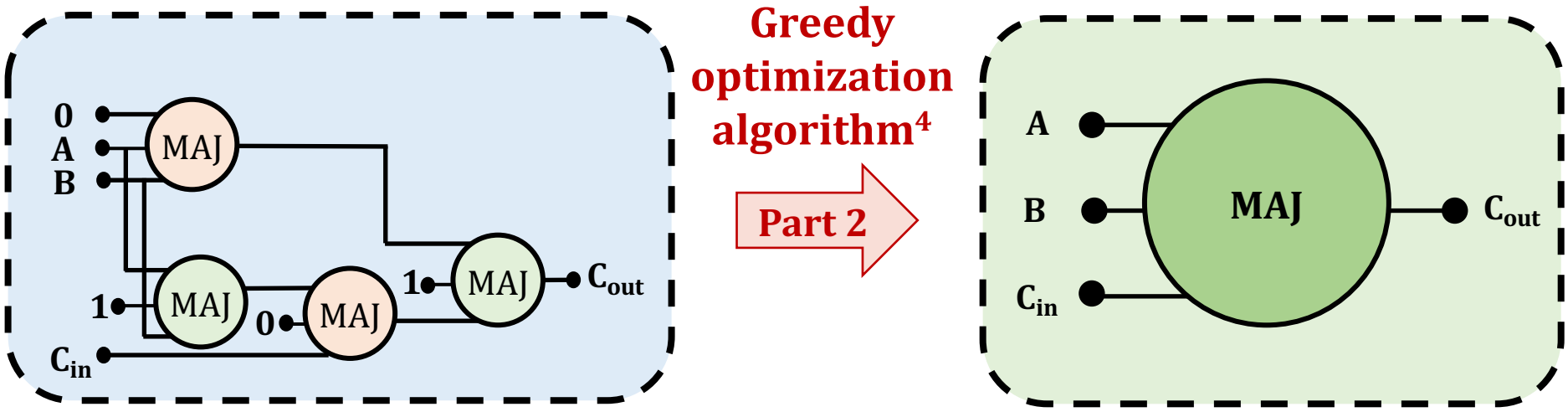


output is “0” only when A = B = “0”



Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit

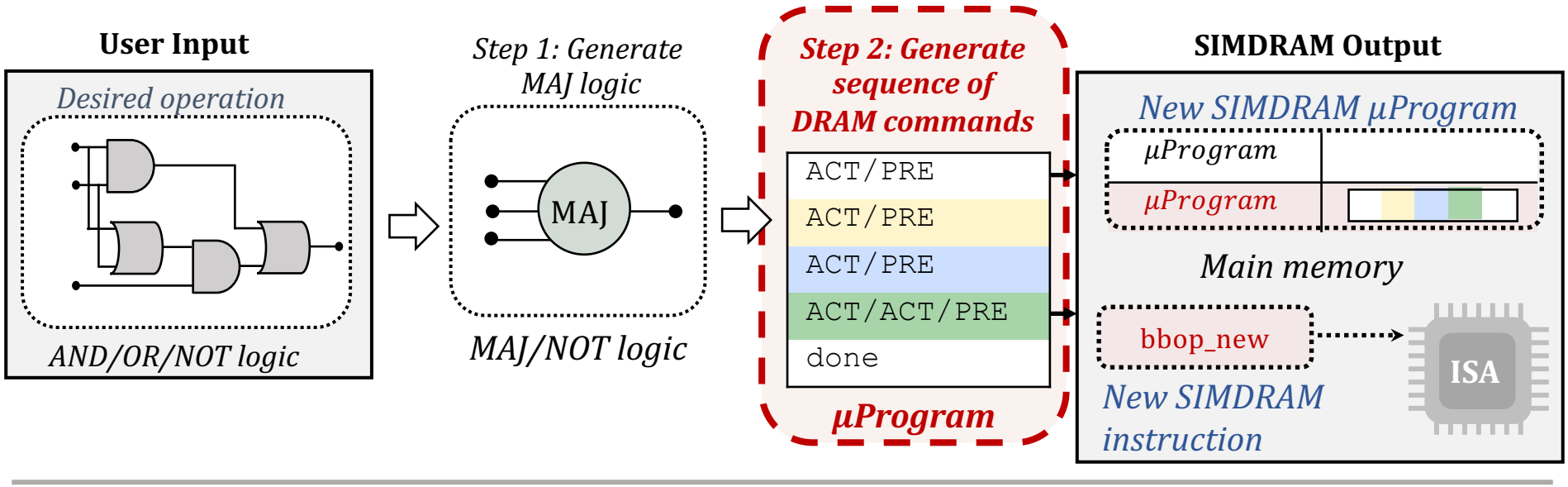
Step 1: Efficient MAJ/NOT Implementation



Step 1 generates an optimized MAJ/NOT-implementation of the desired operation

[4] L. Amarù et al, "*Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization*", DAC, 2014.

SIMDRAM Framework: Step 2



Step 2: µProgram Generation

- **µProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

- **Goal of Step 2**: To generate the µProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate µProgram

Step 2: µProgram Generation

- **µProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

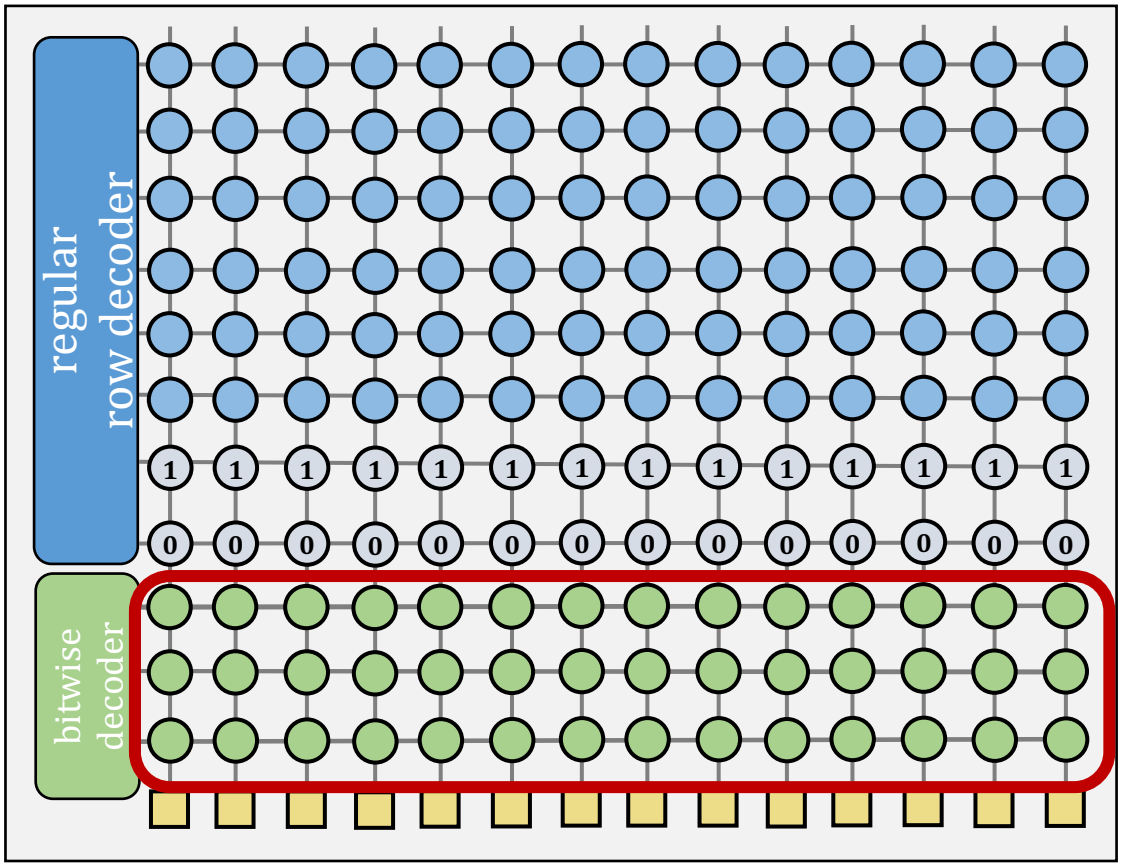
- **Goal of Step 2**: To generate the µProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate µProgram

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM



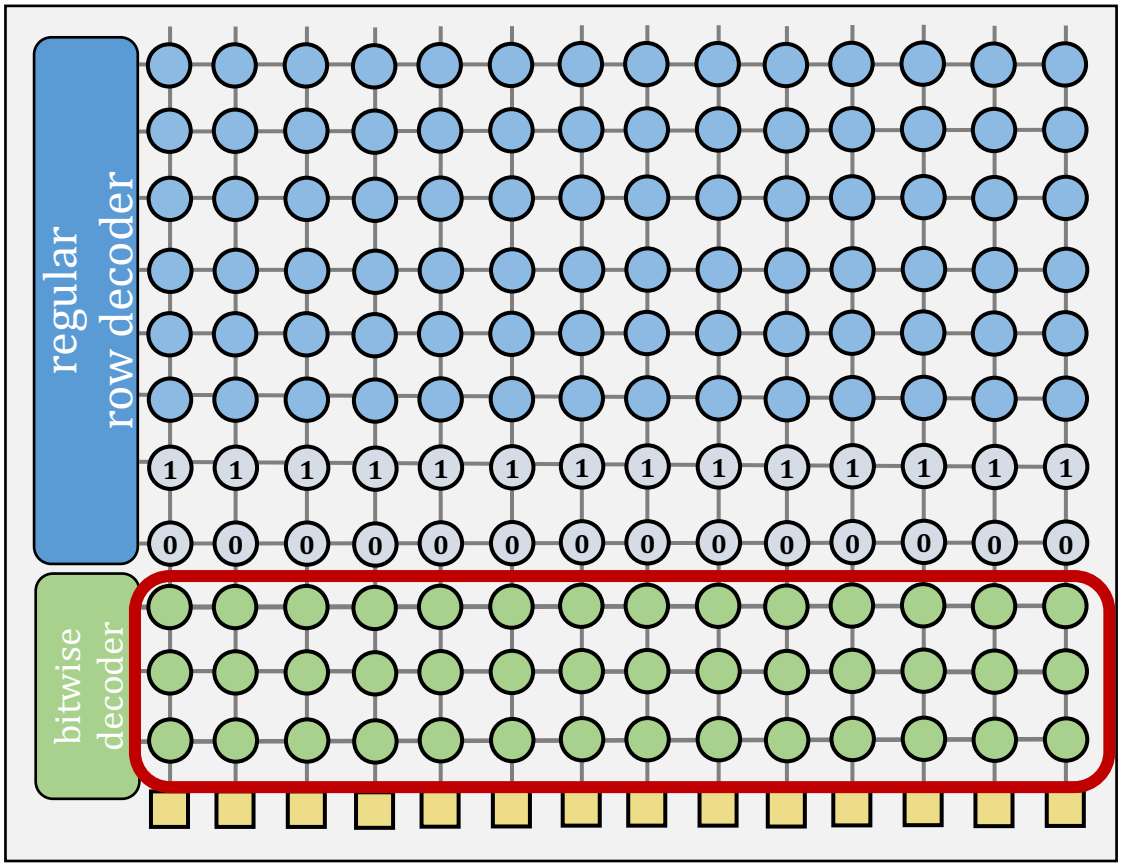
subarray organization

Constraint 1:
Limited number of rows reserved for computation

Compute rows

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM



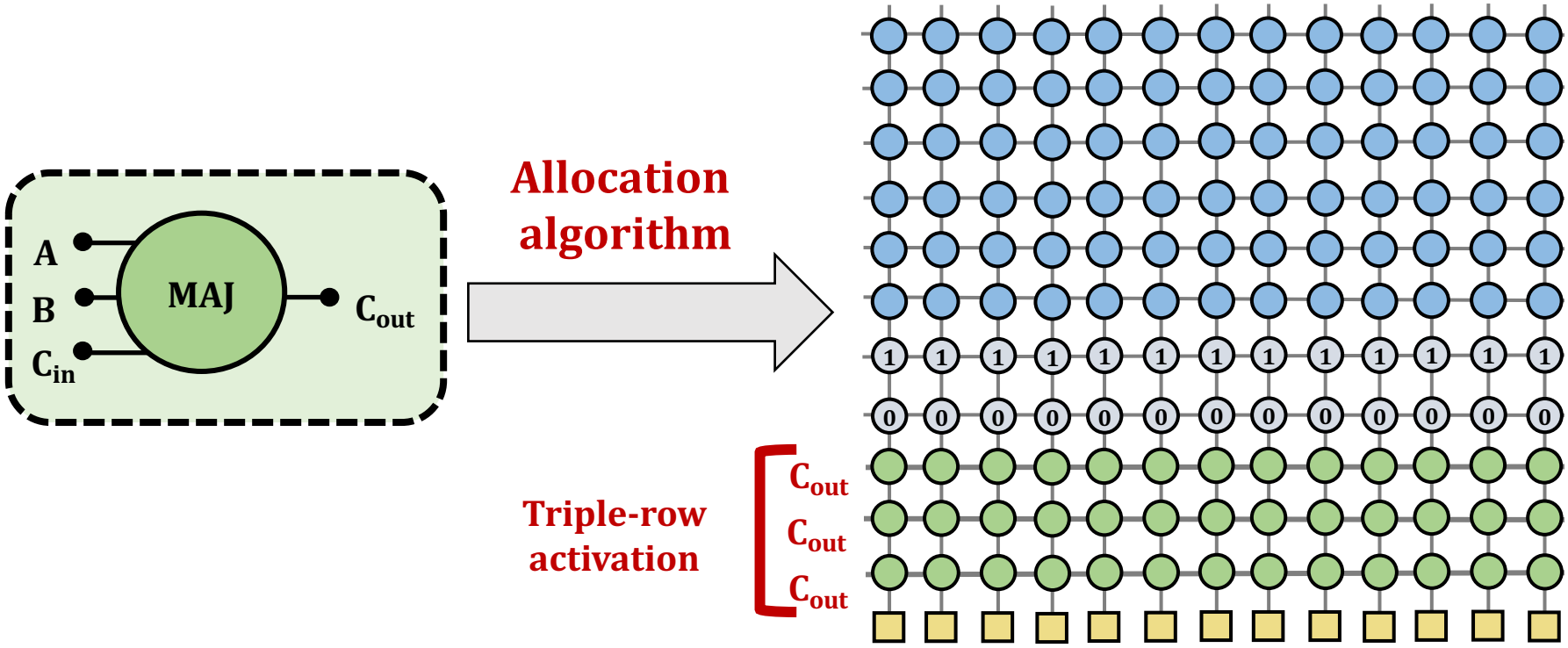
subarray organization

Constraint 2:
Destructive behavior of triple-row activation

Overwritten with MAJ output

Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm:
  - Assigns as many inputs as the number of free compute rows
  - All three input rows contain the MAJ output and can be reused



Step 2: µProgram Generation

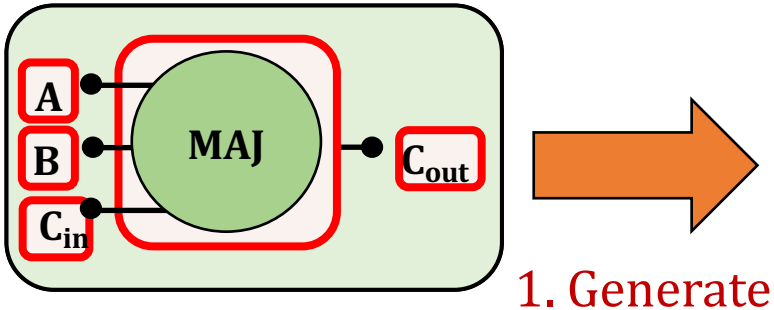
- **µProgram:** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAM uses to execute SIMDRAM operation in DRAM

- **Goal of Step 2**: To generate the µProgram that executes the desired SIMDRAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

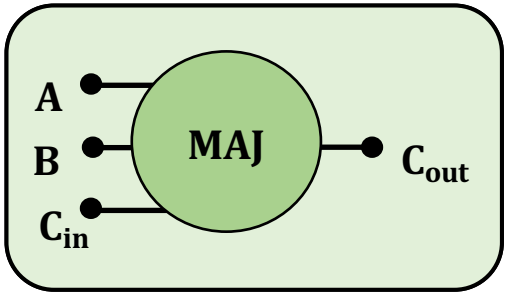
Task 2: Generate µProgram

Task 2: Generate an initial µProgram



1. Generate µProgram

Task 2: Optimize the µProgram



1. Generate µProgram

Initial µProgram

1. Copy A to reserved row **(ACT/ACT/PRE)**

2. Copy B to reserved row **(ACT/ACT/PRE)**

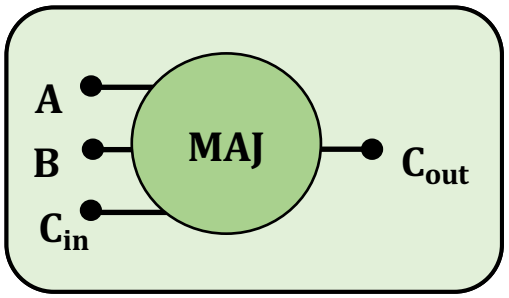
3. Copy C_{in} to reserved row **(ACT/ACT/PRE)**

4. Execute MAJ **(ACT/PRE)**

5. Copy C_{out} to destination row **(ACT/PRE)**

2. Optimize

Task 2: Optimize the µProgram



1. Generate µProgram

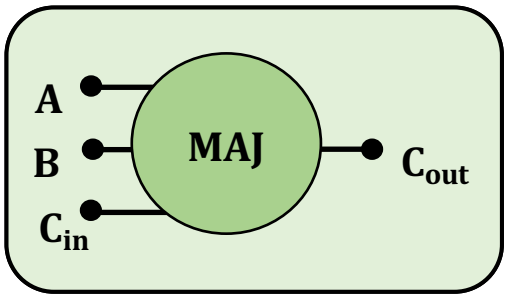
Initial µProgram

1. **Copy** A to reserved row **(ACT/ACT/PRE)**
2. **Copy** B to reserved row **(ACT/ACT/PRE)**
3. **Copy** C_{in} to reserved row **(ACT/ACT/PRE)**
4. Execute MAJ **(ACT/PRE)**
5. Copy C_{out} to destination row **(ACT/PRE)**

Coalesce row copies

2. Optimize

Task 2: Optimize the µProgram



1. Generate µProgram

Initial µProgram

1. Copy A to reserved row **(ACT/ACT/PRE)**

2. Copy B to reserved row **(ACT/ACT/PRE)**

3. Copy C_{in} to reserved row **(ACT/ACT/PRE)**

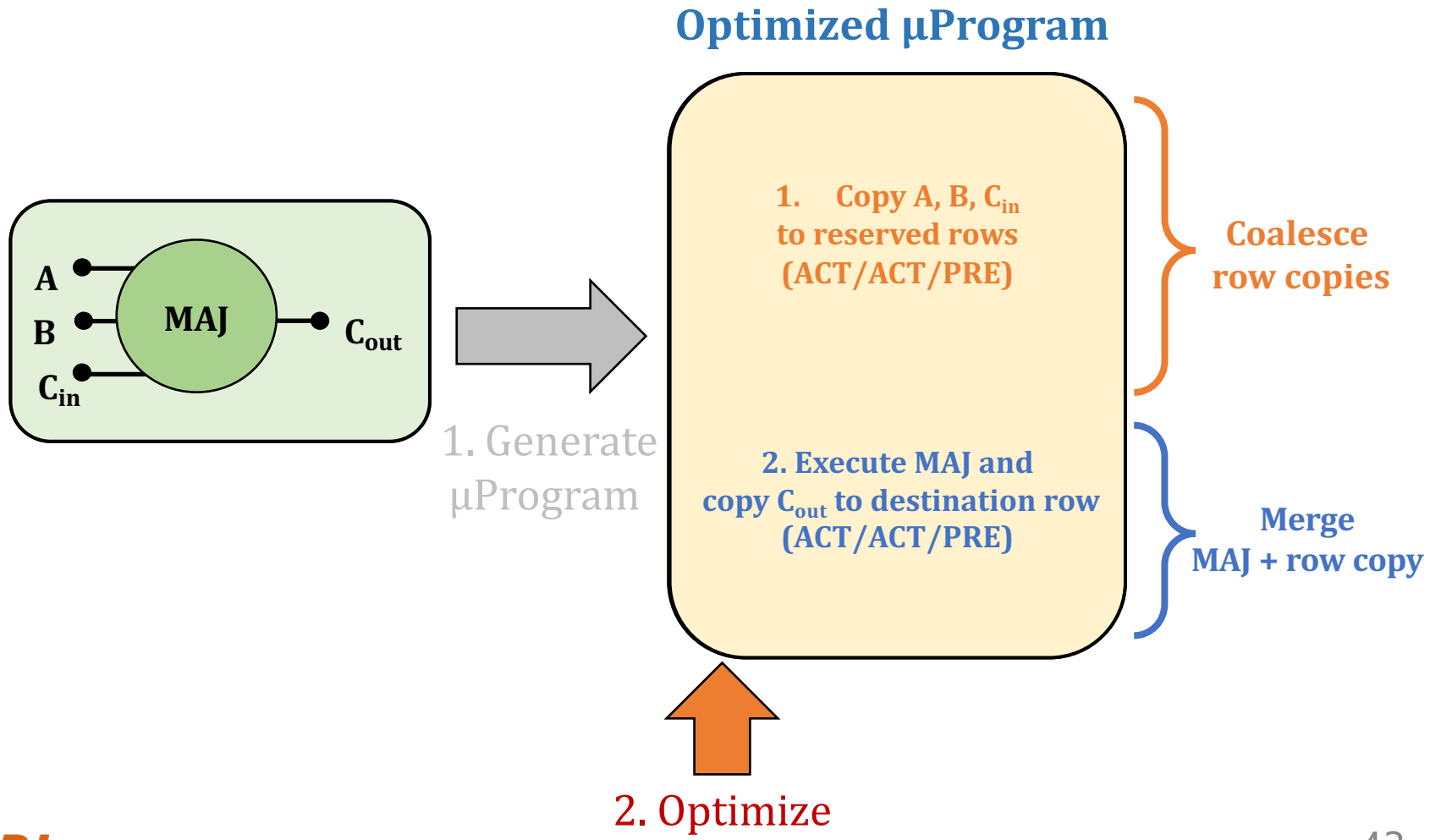
4. Execute **MAJ** **(ACT/PRE)**

5. **Copy** C_{out} to destination row **(ACT/PRE)**

Merge MAJ + row copy

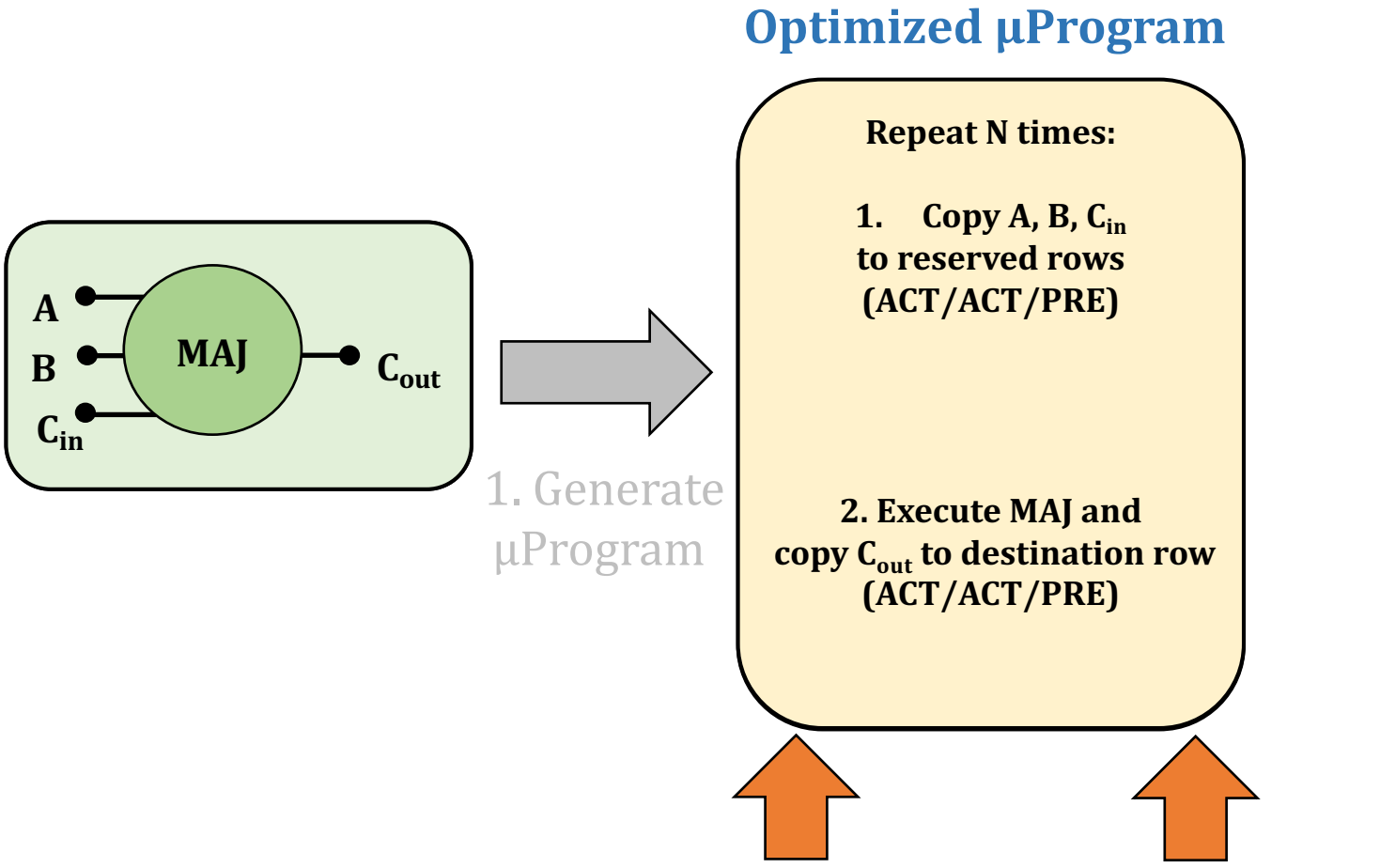
2. Optimize

Task 2: Optimize the µProgram



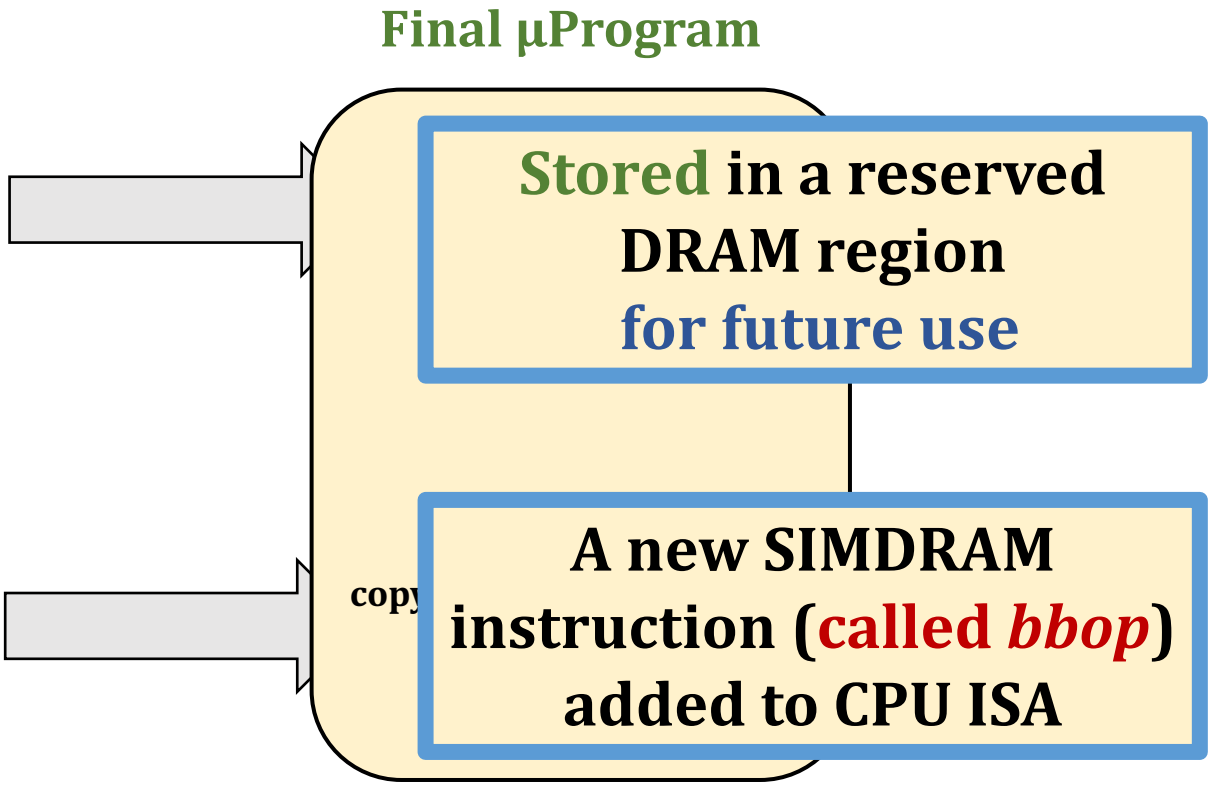
Task 2: Generate N-bit Computation

- **Final μProgram is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion**

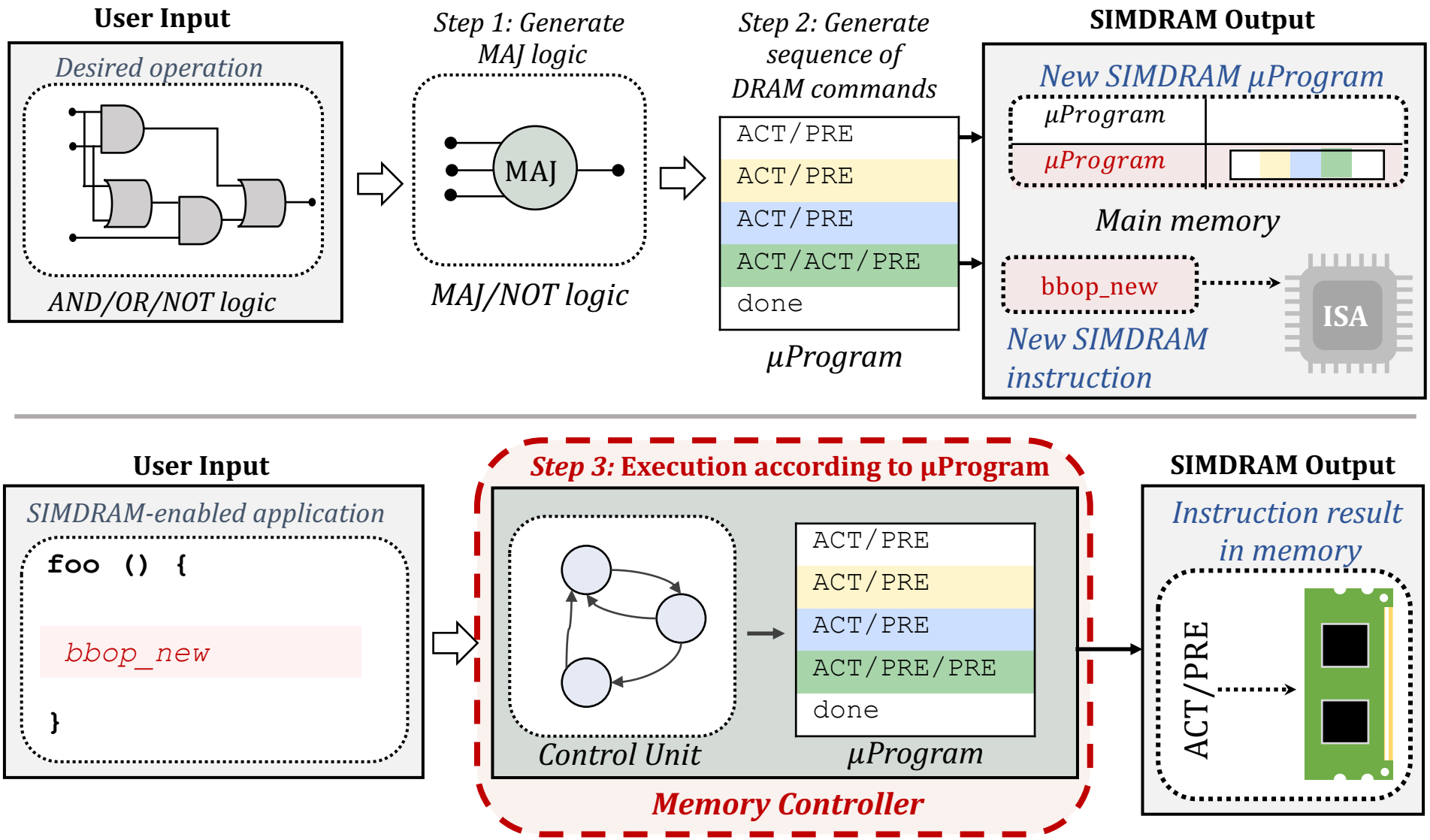


Task 2: Generate µProgram

- **Final µProgram is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion**

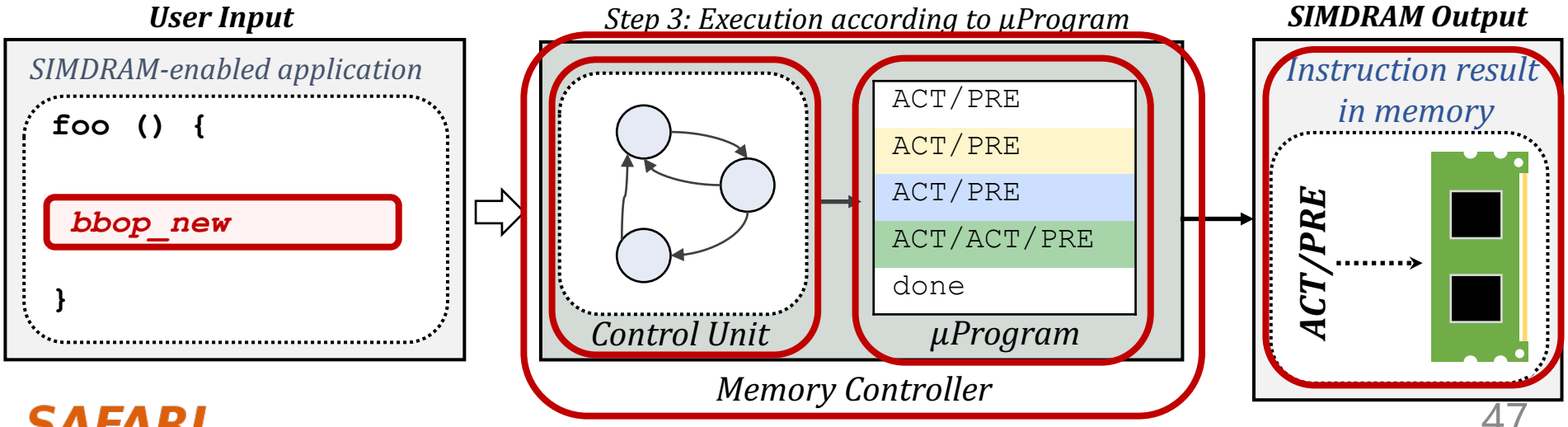


SIMDRAM Framework: Step 3



Step 3: µProgram Execution

- **SIMDRAM control unit:** handles the execution of the µProgram at runtime
- Upon receiving a **bbop instruction**, the control unit:
  1. Loads the µProgram corresponding to SIMDRAM operation
  2. Issues the sequence of DRAM commands (ACT/PRE) stored in the µProgram to SIMDRAM subarrays to perform the in-DRAM operation



Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. **System Integration**
5. Evaluation
6. Conclusion

System Integration

- Efficiently transposing data
- Programming interface
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework

System Integration

Efficiently transposing data

Programming interface

Handling page faults, address translation, coherence, and interrupts

Handling limited subarray size

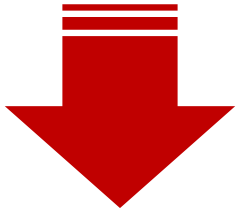
Security implications

Limitations of our framework

Transposing Data

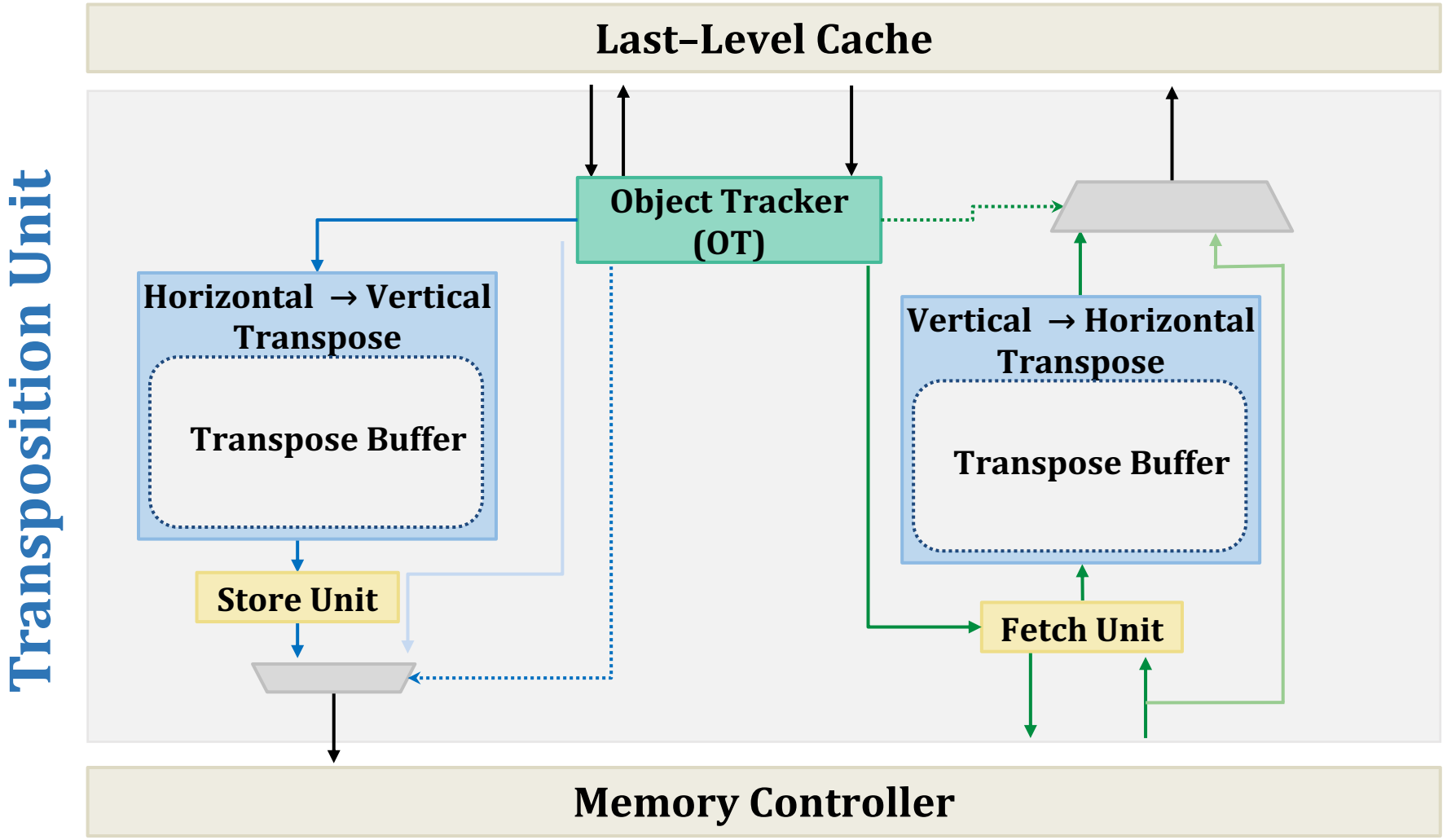
- SIMDRAM operates on vertically-laid-out data

- Other system components expect data to be laid out horizontally

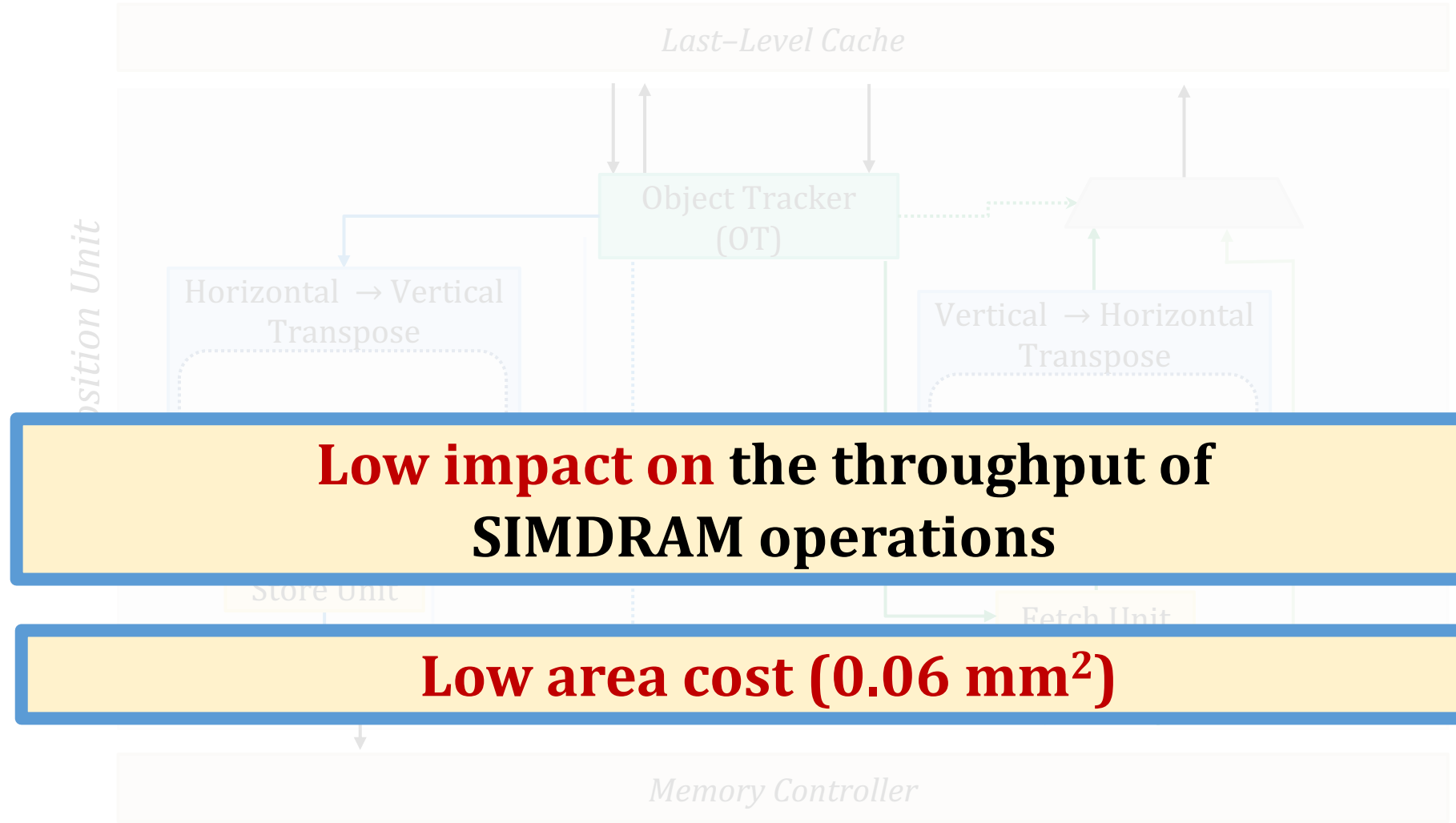


Challenging to share data between SIMDRAM and CPU

Transposition Unit



Efficiently Transposing Data



Low impact on the throughput of SIMDRAM operations

Low area cost (0.06 mm^2)

System Integration

- Efficiently transposing data
- **Programming interface**
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`
2-Input Operation	`bbop_op dst, src_1, src_2, size, n`

Programming Interface

- Four new SIMDRAM ISA extensions

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`
2-Input Operation	`bbop_op dst, src_1, src_2, size, n`
Predication	`bbop_if_else dst, src_1, src_2, select, size, n`

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

Code Using SIMDRAM Instructions

```
1  int size = 65536;
2  int elm_size = sizeof (uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5  …
6  for (int i = 0; i < size ; ++ i){
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAM operations →

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5  bbop_trsp_init(A , size , elm_size);
6  bbop_trsp_init(B , size , elm_size);
7  bbop_trsp_init(C , size , elm_size);
8  uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 …
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```

More in the Paper

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar[1,2] *Geraldo F. Oliveira[1] Sven Gregorio[1] João Dinis Ferreira[1]
Nika Mansouri Ghiasi[1] Minesh Patel[1] Mohammed Alser[1] Saugata Ghose[3]
Juan Gómez-Luna[1] Onur Mutlu[1]

[1]ETH Zürich [2]Simon Fraser University [3]University of Illinois at Urbana–Champaign

coherence, and interrupts

Handling limited subarray size

Security implications

Limitations of our framework

Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

Methodology: Experimental Setup

- **Simulator:** gem5

- **Baselines:**
  - A multi-core CPU (Intel Skylake)
  - A high-end GPU (NVidia Titan V)
  - Ambit: a state-of-the-art in-memory computing mechanism

- **Evaluated SIMDRAM configurations** (all using a DDR4 device):
  - **1-bank:** SIMDRAM exploits 65'536 SIMD lanes (an 8 kB row buffer)
  - **4-banks:** SIMDRAM exploits 262'144 SIMD lanes
  - **16-banks:** SIMDRAM exploits 1'048'576 SIMD lanes

Methodology: Workloads

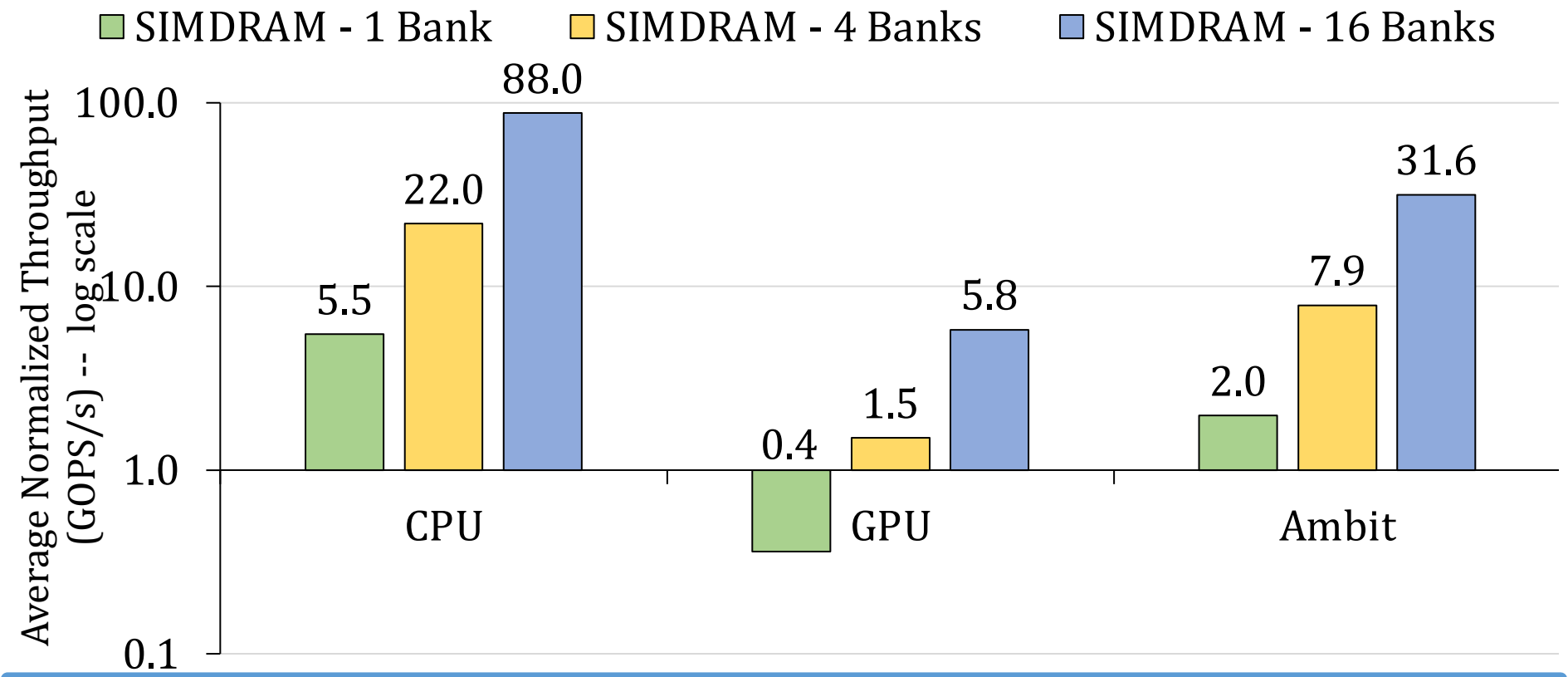
Evaluated:

- 16 complex in-DRAM operations:
  - Absolute
  - Addition/Subtraction
  - BitCount
  - Equality/ Greater/Greater Equal
  - Predication
  - ReLU
  - AND-/OR-/XOR-Reduction
  - Division/Multiplication

- 7 real-world applications
  - BitWeaving (databases)
  - TPH-H (databases)
  - kNN (machine learning)
  - LeNET (Neural Networks)
  - VGG-13/VGG-16 (Neural Networks)
  - brightness (graphics)

Throughput Analysis

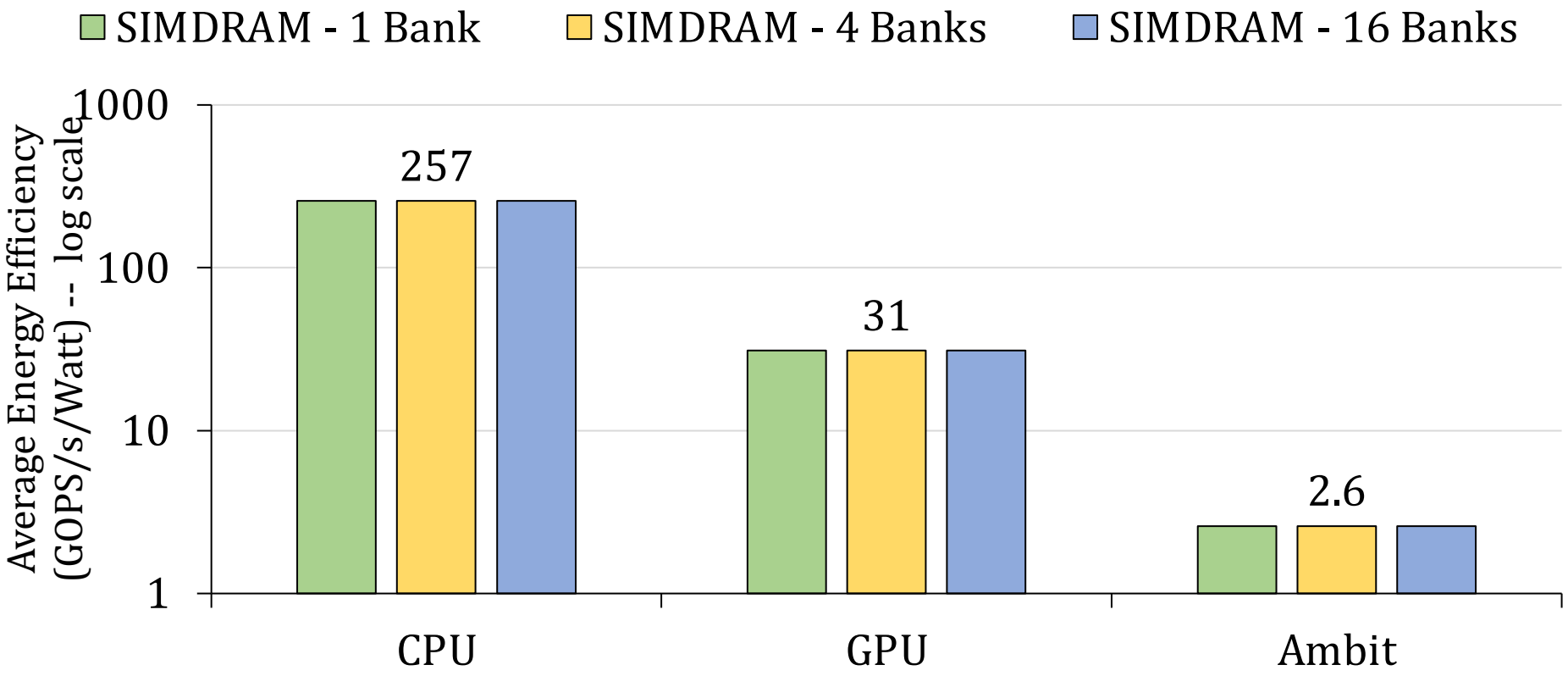
Average normalized throughput across all 16 SIMDRAM operations



SIMDRAM significantly outperforms all state-of-the-art baselines for a wide range of operations

Energy Analysis

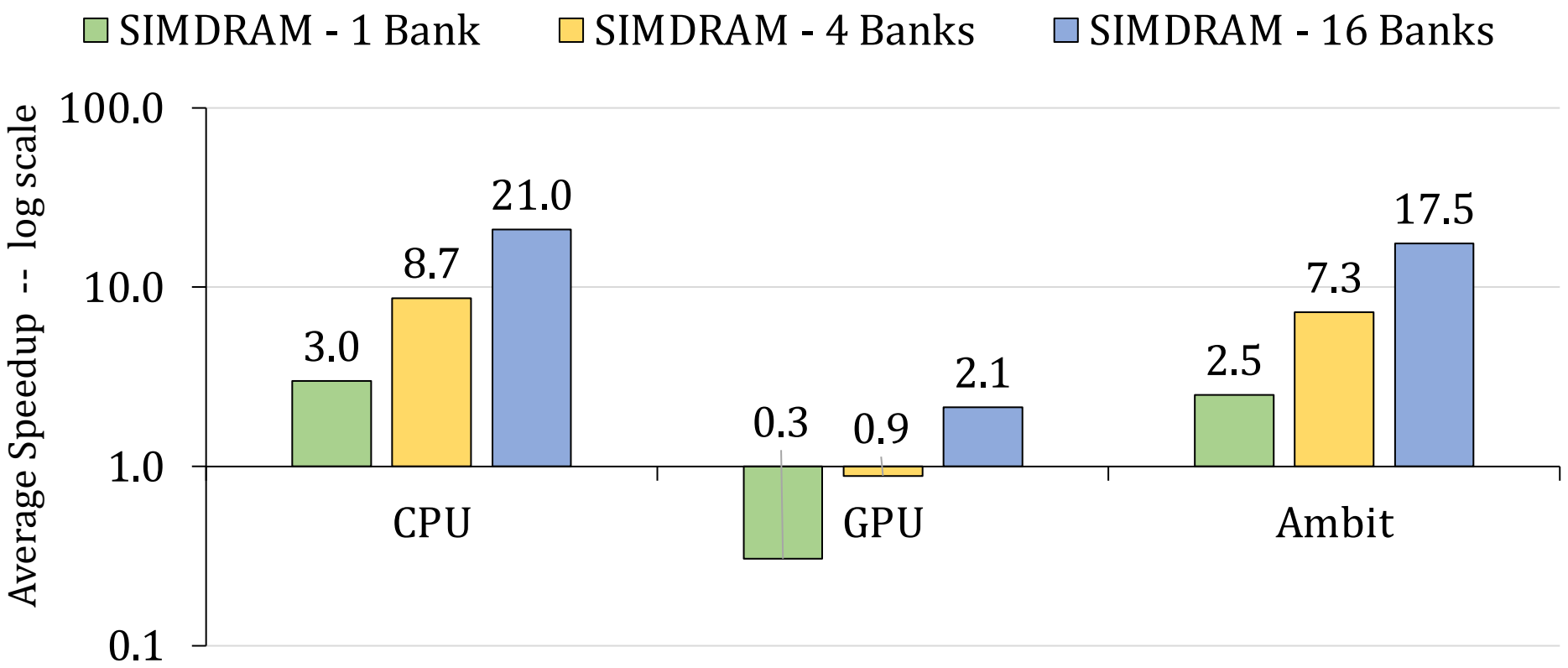
Average normalized energy efficiency across all 16 SIMDRAM operations



SIMDRAM is more energy-efficient than all state-of-the-art baselines for a wide range of operations

Real-World Application

Average speedup across 7 real-world applications



SIMDRAM effectively and efficiently accelerates many commonly-used real-world applications

More in the Paper

- **Evaluation:**
  - Reliability
  - Data movement overhead
  - Data transposition overhead
  - Area overhead
  - Comparison to in-cache computing

More in the Paper

- Evaluation:
  - Reliability
  - Comparison to in-cache computing

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar[1,2] *Geraldo F. Oliveira[1] Sven Gregorio[1] João Dinis Ferreira[1]
Nika Mansouri Ghiasi[1] Minesh Patel[1] Mohammed Alser[1] Saugata Ghose[3]
Juan Gómez-Luna[1] Onur Mutlu[1]

[1]ETH Zürich [2]Simon Fraser University [3]University of Illinois at Urbana–Champaign

Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion

Conclusion

- **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  1. Efficiently computing complex operations
  2. Providing the ability to implement arbitrary operations as required
  3. Using a massively-parallel in-DRAM SIMD substrate

- **Key Results**: SIMDRAM provides:
  - **88x** and **5.8x** the throughput and **257x** and **31x** the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
  - **21x** and **2.1x** the performance of the CPU and GPU over seven real-world applications

- **Conclusion:** SIMDRAM is a promising PuM framework
  - Can ease the adoption of processing-using-DRAM architectures
  - Improve the performance and efficiency of processing-using-DRAM architectures

SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

P&S Processing-in-Memory
Fall 2022
10 January 2023

Nastaran Hajinazar* **Geraldo F. Oliveira***

Sven Gregorio Joao Ferreira Nika Mansouri Ghiasi

Minesh Patel Mohammed Alser Saugata Ghose

Juan Gómez–Luna Onur Mutlu

SAFARI SFU SIMON FRASER UNIVERSITY ETH zürich UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN